Advanced Code Reuse Attacks Against Modern Defences

Advanced Code Reuse Attacks Against Modern Defences

Wang Chenyu

School of Physical and Mathematical Sciences

2019

Wang Chenyu

2019

Advanced Code Reuse Attacks Against Modern Defences

Wang Chenyu

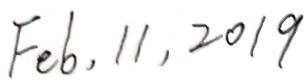
School of Physical and Mathematical Sciences

A thesis submitted to Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.



. . . . . .

Date

. . . . . .

Signature

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

12 Feb 2019 . . . . . .

Date

. . . . . .

Signature

Authorship Attribution Statement

This thesis contains materials from 2 papers published in the following peer-reviewed journals where I was the first and/or corresponding author.

Chapter 3 is published as Chenyu Wang, Tao Huang, Hongjun Wu, On the Weakness of Constant Blinding PRNG in Flash Player, The 20th International Conference on Information and Communication Security. ICICS 2018, LNCS 11149, 1-17, 2018

- A/Prof Wu provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by Dr Huang.
- I studied the constant blinding mechanism in JIT Compiler in Flash Player.
- I analysed the weakness in pseudo random number generator (PRNG) in Flash Player. A/Prof Wu assisted me in cryptographic analysis on the PRNG.
- I wrote one proof-of-concept exploit to support our attack on the PRNG.
- All pictures, tables, and data evaluations were conducted by me.

Chapter 5 is published as Chenyu Wang, Bihuan Chen, Yang Liu, Hongjun Wu, Layered Object-Oriented Programming: Advanced VTable Reuse Attacks on Binary-Level Defense, IEEE Transactions on Information Forensics and Security. 14(3): 693-708, 2019.

- A/Prof Wu provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by Dr Chen and A/Prof Liu.
- I discussed the basic information of modern control flow integrity enforcement and generalized their enforcement policy.
- I analysed the weakness in target mitigations and developed tools to support our argument.

- I wrote proof-of-concept exploits to demonstrate our attack on complicated software.
- All pictures, tables, and data evaluations were conducted by me.

Feb. 11, 2019

. . . .

Date

. . . .

Signature

Acknowledgments

I would like to take this as an opportunity to thank everyone who has supported me in this research during the last couple of years. All of your help, including academic research and everyday support, have made my time as a PhD student valuable treasure in my life.

First of all, I would like to thank Prof. Hongjun Wu for being an excellent supervisor to me. Your insight in the area of security research and your guidance in our weekly group meeting are the cornerstone of my research on exploit development. In addition, your insistence on pursuing the most challenging topics in this area drives me to follow the top research work and engineering practice in the world. Thank you for giving me the time to write a thesis that I am actually happy with.

I would also like to thank Prof. Yang Liu for taking the time to read and revise my papers. Given your valuable experience in applied computer security and software engineering, I am able to evaluate my exploitation techniques in a formal and systematic approach. With your financial support during the last year, I can focus on my research work and have my papers published.

Furthermore, I would also like to express my gratitude for my colleagues Tao Huang and Bihuan Chen. I am also grateful for their contribution for refining my research paper. I learn a lot from them on how to prepare and organize a high-quality research paper.

I would like to thank the anonymous examiners of this thesis for their valuable time spent and the comments they provide.

Finally, I express my gratefulness to my parents during the whole time of my PhD study. Their firm support and encouragement empower me to overcome the obstacles during those years. I also acknowledge the funding support from the Research Scholarship from Singapore government.

List of Works

[1] Chenyu Wang, Bihuan Chen, Yang Liu and Hongjun Wu. Layered Object Oriented Programming: Advanced VTable Reuse Attacks on Binary-Level Defense. In *Transactions in Information Forensics and Security (TIFS)*, to appear, 2018.

[2] Chenyu Wang, Tao Huang, and Hongjun Wu. On the Weakness of Constant Blinding PRNG in Flash Player. In *20th International Conference on Information and Communications Security (ICICS 2018)*, to appear, 2018.

[3] Chenyu Wang, and Hongjun Wu. CVE-2017-3000: A random number generator vulnerability used for constant blinding that could lead to information disclosure https://helpx.adobe.com/security/products/flash-player/apsb17-07.html

Contents

Abstract

Exploit development is an arm race between attackers and defenders. In this thesis, I will introduce the development of code reuse attacks in recent years together with control flow integrity (CFI).

I will give a deep insight in the CFI based on the binary code and demonstrate how limited those mitigations are against sophisticated code reuse attacks. TypeArmor and vfGuard are believed to be sufficient in defending against vtable reuse attacks. Both techniques use semantic information as the control flow integrity enforcement policy. We propose Layered Object-Oriented Programming (LOOP), an advanced vtable reuse attack, to show that the coarse-grained CFI strategies are still vulnerable to vtable reuse attacks. In LOOP, we introduce argument expansion gadgets and transfer gadgets to respectively bypass TypeArmor and vfGuard. We generalize the characteristics of both gadgets, and develop a tool to discover them at binary level. We demonstrated that under the protection of TypeArmor and vfGuard, Firefox, Adobe Flash Player and Internet Explorer are all vulnerable to LOOP attacks. Furthermore, we evaluate the availability and complexity of both gadgets in common software or libraries.

Moreover, we will explain what is JIT spray attack and how constant blinding is expected to defend against such attack. We study the design and implementation of constant blinding mechanism in Flash Player and analyse the weakness in its pseudo random number generator (PRNG). Such weakness can be exploited to recover the seed value in PRNG, thus weakening the constant blinding in Flash Player. We propose two methods to circumvent constant blinding in Flash Player and demonstrate that these two methods are both practical via presenting proof-of-concept attacks based on existing vulnerability. We have reported the issue to Adobe Flash security team and CVE-2017-3000 is assigned to us. Furthermore, we implement a prototype tool Constant Blinding Enhancement (ConBE) based on dynamic instrumentation framework to defend against our proposed attacks. In ConBE, we provide a stronger

defence than the official patch of Flash Player.

We also study the JIT engine in Edge and Chrome browsers and try to discover the non-blinded constant in the JIT code. We propose Blockade, a grammar-based fuzzing framework, to search for cases where constant numbers are not blinded (non-blinded constant) in JIT code. We revisit the grammar of JavaScript and discover that proper grammar combined with efficient generation policy can greatly help us dig for the non-blinded constant in JIT code. Our work shows that structural information in script language can be utilized to release non-blinded constant number. We run Blockade on Microsoft Edge and Google Chrome. The result shows that in addition to the cases that have been discovered in previous work, our tool is able to find more cases of non-blinded constant. We find that array offset, object field, global variable and even number of statements in script can be used to emit non-blinded constant in JIT code.

Chapter 1

Introduction

With the increasing reliance on the computer today, computer security has become a growing concern to the public. In recent years, severe vulnerabilities are reported in media attracting the public attention. In 2014, HeartBleed vulnerability [19] was discovered in the widely used cryptographic library OpenSSL. In a single HeartBleed attack, attackers can leak at most 64 kilobytes from memory, including the password or other senstive information of users. In 2015, two zero-day vulnerabilities (CVE-2015-5119 [8] and CVE-2015-5122 [9]) in Flash Player were leaked from HackingTeam [13]. The event rings the alarm bell to the public that the unknown zero-day vulnerability is threatening everyone's life. In 2016, DirtyCow vulnerability [6] in Linux kernel was reported. A race condition in Linux kernel's memory subsystem allows a local unprivileged user to gain write access to otherwise read-only memory mappings and thus escalate their privileges on the system. In 2017, the outbreak of EnternalBlue [26–30] and its descending variant WannaCry hit multiple infrastructures in Australia, United Kingdom, Spain and the United States.

During those years, high-risk zero-day vulnerabilities are sold for hundreds of thousands of dollars [14]. In the famous hacking competition Pwn2Own [36] and Mobile Pwn2Own [35], a successful exploit can earn a reward between 35K and 100K US dollars based on the severity of the exploit on different platforms [39]. Various bug

bounty programmes, e.g. HackerOne [18] and iDefence [21], also drive more security researchers to dig deeper in this area and discover more high-risk vulnerabilities.

With so much attention and profit in computer security, the defence against a successful exploit has become a hot topic. In software security, various kinds of mitigation techniques were proposed in industry and academic work. For example, Control Flow Integrity [40] was first proposed in 2005. Now similar techniques have been deployed in the widely used software, such as CFG in Microsoft Edge [24], LLVM-CFI in Android [3] and kCFI in Linux Kernel [94].

However, the competition between attackers and defenders is still going on. In recent years, novel code reuse attacks, like Counterfeit Object Oriented Programming (COOP) [108] and variations of Return Orinted Programming (ROP) [48, 55], are proposed to defeat the state-of-the-art defence mechanisms. Analysing the weakness in defence mechanism usually requires a deep understanding of the vulnerability root cause, good skills on reverse engineering and a huge accumulation of knowledge on each module in computer system.

This thesis will present some new code reuse attacks developed in our research. We demonstrate a variant of COOP, Layered Object Oriented Programming (LOOP), to demonstrate the weakness in some binary based mitigation mechanisms. Our work shows that the current defence based on information extracted from binary code are not sufficient to defend against sophisticated code reuse attacks. Moreover, we demonstrate that the JIT spray attack, a technique to generate ROP gadgets, is still possible in Flash Player even if the defence is turned on. We also study Microsoft Edge and Google Chrome to find non-blinded constants in JIT code, which can be used as ROP gadgets.

We will give definitions on some key concepts used in this thesis in the following sections of this chapter, then we will introduce the structure of the thesis.

1.1 Vulnerability

NIST Special Publication 800-53 Rev 4 [52] defines a vulnerability as *Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.*

In this thesis, we focus on vulnerabilities in software. Such vulnerabilities are usually triggered by some malformed input from attackers. However, the consequences of the vulnerabilities are not clear in the definition. To give a better view of such vulnerabilities, we categorize the potential consequence of a vulnerability into four types.

- **Code Execution** An attacker provides malformed input data to be processed by the victim program and executes arbitrary code in the context of the program. For example, CVE-2015-5119 and CVE-2015-5122 allow attackers to execute arbitrary code via malicious script.

- **Privilege Escalation** An attacker exploits the logic error or memory corruption in victim program to perform privileged operations, which should be denied according to the security policy designed in the system. Sometimes, privilege escalation does not require the ability to execute arbitrary code in the context of victim program. Therefore we put privilege escalation as a different category as code execution. For example, DirtyCow (CVE-2016-5195) exploits a logic error in memory module to escalate privilege.

- **Information Disclosure** An attacker can read information in the system, which is not intended to be exposed to him. For example, attackers can obtain the pointer value in JavaScript language, where the pointer of any object should never be disclosed to user. In the case of HeartBlead, an attacker is able to read far more data than expected from memory of a remote server.

- **Denial of Service** An attacker can terminate the execution of a system. In the context of Linux Kernel, attackers can crash the kernel via a system call

combined with a malformed input data.

In this thesis, we will mainly focus on the vulnerabilities that result in code execution, and hence, the defence mechanisms that are designed to protect victim program from arbitrary code execution.

1.2 Exploit

In NICCS glossary provided by the Department of Homeland Security of the United States [12], exploit is defined as *A technique to breach the security of a network or information system in violation of security policy.* In the context of software security, we are mainly talking about the exploits that aim to achieve arbitrary code execution in common software systems.

Trend Micro [15] concludes that a successful and high-risk exploit should consist of an arbitrary write primitive. Among multiple exploitation techniques, the write primitive is still a cornerstone of a successful exploit. To be more specific, the first step after triggering the vulnerability is usually to gain the write primitive in the memory space of victim application. Such primitive enables an attacker to perform more operations for further exploitation. For example, attackers can corrupt data related with control flow to achieve control flow hijacking, or attackers can corrupt data related with user credential to achieve privilege escalation.

Besides write primitive, Heap Spray [73, 105, 116, 129, 130] techniques are widely used in recent years to increase the success rate of an exploit. Generally speaking, code that sprays the heap attempts to put a certain sequence of bytes at a given address in memory of target application by allocating large chunks of memory heap and filling the chunks with desired bytes. The advantage of heap spray is that attackers can always put some predetermined values at a given address weakening the protection provided by some randomization techniques.

1.3 Mitigation

In computer security, mitigation (or risk mitigation) is used to rephrase the defence technique. Mitigation is defined as *application of measure or measures to reduce the likelihood of an unwanted occurrence and/or its consequences* [63]. We should note that the mitigation only raises the bar of exploit development and aims to prevent attackers from writing successful exploits. However, no mitigation can cover all kinds of exploit techniques in the world. The combination of various mitigation mechanisms assure the software security today.

After multiple years of contribution from the security community, many mitigation mechanisms have been deployed. In this section, I will introduce some of the most common mitigations in software security.

- **ASLR** Address Space Layout Randomization [31,38] is a mitigation technique, which randomizes the base address of loaded module and allocated heap. With ASLR, attackers cannot use fixed values to corrupt critical data in memory.

- **DEP** Data Execution Prevention [5] is used to enforce **W**$\oplus$**X** policy in memory space, i.e., the memory space cannot be executable or writable at the same time. Even if attackers can write shellcode directly into a writable region, attackers still cannot execute those shellcode because the region is not executable.

- **Stack Cookie** Stack Cookie is a compiler option provided by mainstream compilers today [16,17]. Extra canary cookie is inserted between the return address and local variable on stack. During the epilogue of function, this cookie is compared against the global master cookie. If values are different, the program is terminated to prevent potential buffer overflow attacks.

- **SafeSEH** Safe Structured Exception Handler [33] is another compiler option on Windows platform. The binary modules compiled with this flag will maintain a whitelist of exception handler functions. SafeSEH aims to block the control

flow hijacking from exception to illegal address in memory.

In this thesis, we will introduce more modern mitigation techniques, which are proposed in recent years. We further demonstrate more sophisticated exploit techniques to prove the weakness in those mitigation techniques.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 provides the technical background of code reuse attacks and corresponding mitigations. We will first present the basic ideas of code reuse attacks, including VTable reuse attacks and JIT spray attacks. On the side of mitigation, we will discuss how CFI (control flow integrity) and constant blind are expected to mitigate those attacks respectively.

Chapter 3 deals with weakness in constant blinding PRNG in Flash Player. Under a common attacking model today, we show that the design and implementation of the PRNG enable an attacker to recover the seed with little effort for further JIT spray attacks. Moreover, we implement and evaluate a mitigation prototype against our proposed attacks.

Chapter 4 illustrates our study on discovering non-blinded constants in Microsoft Edge and Google Chrome. Our work shows that the structural information hidden in script language can be served as implicit constant number in JIT code. We further propose a fuzzing-based framework to search for cases of non-blinded constants.

Chapter 5 evaluates two state-of-the-art binary level mitigations, TypeArmor and VFGuard, against VTable reuse attacks. We demonstrate a novel VTable reuse attack, Layered Object Oriented Programming, to prove the weakness in both designs.

Chapter 6 summarizes the work in this thesis and draws conclusions. We also discuss possible directions in future research.

Chapter 2

Modern Attack and Mitigation

2.1 Return-Oriented-Programming

2.1.1 Introduction

Since executing shellcode in non-executable memory is forbidden due to the existence of DEP, attackers turned their attention to code reuse attacks, i.e. using the existing code in static libraries or dynamically generated code to launch attacks. ROP attack is one typical kind of code reuse attacks.

ROP gadget refers to a short sequence of assembly instructions that end with *ret*. In ROP attack [104, 120], attackers usually have to prepare addresses of gadget in stack and chains those ROP gadgets via *ret* instruction. In a chain of ROP gadgets in Figure 2.1, gadgets are supposed to achieve various kinds of operations, e.g. (G1) load value from stack into register, (G2) arithmetic operation, (G3) store value into memory, (G4) load value from memory into register, (G5) invoke function and etc. Attackers can combine different types of the gadgets to launch the attack.

2.1.2 Advanced ROP Attacks

In recent years, more advanced ROP attacks were proposed.

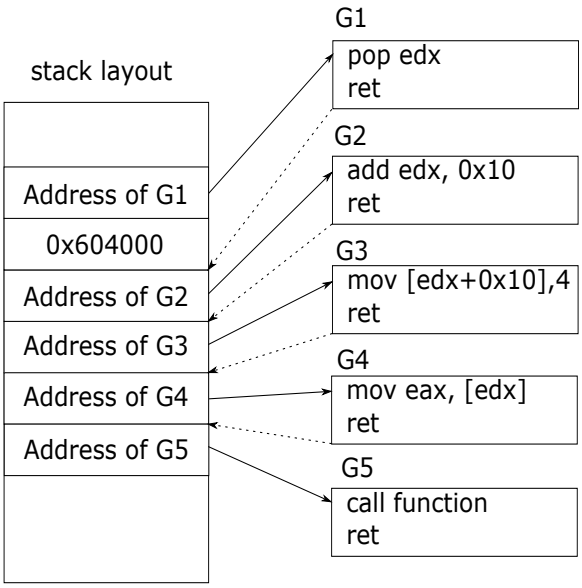


Figure 2.1: Stack Layout in ROP

- **JIT ROP** Strictly speaking, JIT-ROP [113] does not use dynamically generated code as ROP gadget. Instead, JIT-ROP discusses how to disclose critical information from memory and construct the gadgets on-the-fly.

- **Blind ROP** BROP [48] provides attackers to construct ROP attack in a remote server application even if without the knowledge of the target binary. Attackers can identify different types gadgets for constructing exploit step by step. The hidden assumption in BROP is that the memory layout of target process will not change after crash.

- **Position Independent ROP** PIROP [79] provides attackers another way to target remote server application even if memory space layout of target process will change after crash. In PIROP, attackers first construct ROP gadgets with a few-bytes data corruption without crashing the victim application and then leaks critical information defeating ASLR. The hidden assumption in PIROP is that the attacker has knowledge of the target application.

```
script code:
fun() {
      a = 0xc3585a59;
      a = 0xdeadbeef;
}

JIT code without constant blinding:
mov     ecx, 0xc3585a59
mov     dword ptr [edx+0x14],ecx
mov     ecx, 0xdeadbeef
mov     dword ptr [edx+0x14],ecx

JIT code with constant blinding:
mov     ecx, 0xccf1d312
xor     ecx, 0xfa9894b
mov     dword ptr [edx+0x14],ecx
mov     ecx, 0xd10437a4
xor     ecx, 0xfa9894b
mov     dword ptr [edx+0x14],ecx
```

Listing 2.1: Constant Blinding in JIT Code of Flash Player

2.2 Constant Blinding and JIT Spray Attack

2.2.1 JIT Spray Attack

CFI and G-free [99] try to reduce the available gadgets for code reuse attacks in static code. However, JIT spray attack uses the dynamically generated code to provide ROP gadgets. In another word, JIT spray attack is out of the protection scope of those mitigations. In the work of Blazakis [49] and Sintsov [112], the constant number in JIT-generated code provides a 4-byte long gadget for attackers in executable code region.

In JIT Spray attacks, the constant in script can be used as a short gadget for further exploitation. Constant number `0xc3585a59` in Listing 2.1 will be emitted into code heap by JIT compiler as an immediate value in instruction *mov ecx*, `0xc3585a59`. The byte sequence of this instruction will be `0xb9 0x59 0x5a 0x58 0xc3` in memory on a little-endian system. If the sequence of bytes is located at `0x602010` and attacker

hijacks the control flow to `0x602011`, the misaligned byte sequence will be interpreted as *pop ecx* (`0x59`), *pop edx* (`0x5a`), *pop eax* (`0x58`) and *ret* (`0xc3`) from the view of assembly code and be used as an ROP gadget.

2.2.2 Constant Blinding

Constant blinding was proposed to mitigate JIT-spray attack in JITSafe [57] and has already been deployed in the script engine that supports JIT compilation. To be more specific, it prevents the value of a constant number appearing in memory. The most common solution is to generate a secret cookie and XOR the constant number with the secret cookie to get a new number. At the moment of emitting JIT code, the new number will be moved into a register first. The register will be then XORed with the secret cookie to restore the original number. We give an example in Listing 2.1 to demonstrate how constant blinding works. The constant number, `0xc3585a59` was blinded by a secret cookie `0xfa9894b`. In the following of this thesis, we call the generated random number as secret cookie and call the value, which results from XOR operation on original value and secret cookie, as blinded constant.

For a user-defined function, the JIT code of the function is generated at the first time the function is being called [22]. The JIT compiler will check if JIT code of this function exists. If the code has not been generated yet, the JIT compiler will generate the code first and execute the generated code. Otherwise, the JIT code will be executed directly. Before JIT compilation, constant values in script are stored in a symbol table located in memory region with readable and writeable permission. We will discuss how to exploit the JIT compilation process to launch our attack later in Section 3.3.4.

2.3 Control Flow Integrity and VTable Reuse Attack

2.3.1 Introduction

In 2005, Abadi et al. proposed Control Flow Integrity (CFI) [40] enforcement to defend against ROP attacks. Now there are various implementations of CFI introduced in academia and industry. CFI aims at enforcing the integrity of the targets of indirect control-flow transfers, including indirect jumps, indirect calls and returns. By assigning signatures to indirect call/jump targets and instrumenting signature checks before indirect jump/call instructions, CFI is expected to stop the unexpected control-flow transfers. Precise or fine-grained CFI assigns a unique signature for every indirect transfer target, but introduces significant overhead in practice. This motivates the development of imprecise or coarse-grained CFI that assigns a limited number of signatures for protected targets [60, 100, 134]. However, many of these defenses have been already bypassed [55, 69, 74, 78].

2.3.2 VTable Reuse Attack

Virtual Function Call. C++ introduces polymorphism in its design. When a class declares a virtual function, each object of this class will contain a virtual table (vtable) pointer in memory, which points to a virtual table. A call to a virtual function is usually divided into four steps: (1) argument preparation, (2) vtable pointer fetch, (3) virtual function dispatch, and (4) virtual function invocation. Fig. 2.2 shows the process of a virtual function call. Register `rcx` stores the implicit `this` pointer. Step 1 (the first two `mov` instructions) sets argument registers. Step 2 fetches the vtable pointer stored at the top of the object. Step 3 fetches the function pointer from the virtual table, while Step 4 calls the target function `Afun`. Sometimes, Step 3 and 4 are combined together as `call [rax+offset]`.

The procedure above involves two concepts that will be used in Chapter 5: *dispatch offset* and *vtable offset*. We use *dispatch offset* to represent the offset used in

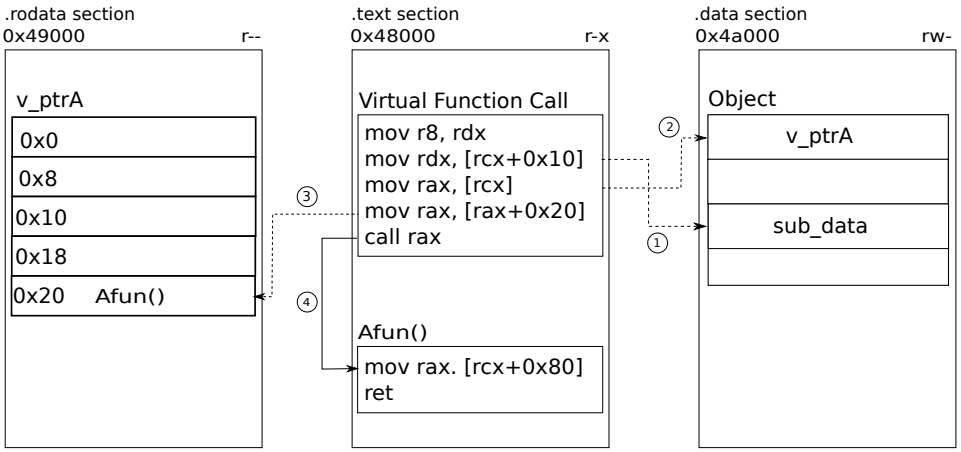


Figure 2.2: Virtual Function Call on Windows (The Dashed Line Denotes Data Fetch and the Solid Line Denotes Control Flow Transfer.)

assembly code for virtual function dispatch and *vtable offset* to refer to the offset of a function pointer to the top of a virtual table. *Dispatch offset* is defined from the perspective of assembly code and *vtable offset* is defined from the perspective of memory layout. In Fig 2.2, `0x20` is the dispatch offset in Step 3 and the vtable offset used to fetch function pointer from virtual table. In Fig. 5.1, the dispatch offset of virtual callsite is `0x18` but the vtable offset of the target function pointer is `0x28`.

The most common exploit on virtual function is vtable injection attack. An attacker overwrites the vtable pointer in victim object, and makes it point to an injected vtable. Thus, in virtual function dispatch, a function pointer under attacker's control is fetched and the control flow is hijacked to any function or gadget in code. However, vtable integrity enforcement techniques [76, 133] make vtable injection attacks less practical.

COOP. Different from vtable injection attacks, vtable reuse attacks, exemplified by COOP [108], do not overwrite the vtable pointer to point to an injected virtual table. Instead, the attacker attempts to overwrite the vtable pointer to point to an existing vtable, and diverts the control flow to an existing virtual function. In particular, to launch COOP attacks, the attacker needs a dispatcher gadget to chain

multiple virtual function gadgets together. Usually the dispatcher gadget is a loop gadget, which iterates over the objects in an array and calls virtual functions one by one. The attacker uses different types of virtual function gadgets in the victim binary to achieve specific operations, e.g., arithmetic calculation and register loading. As COOP relies on existing vtables, it can bypass most CFI enforcement [93, 134] and vtable integrity enforcement [76, 133].

Chapter 3

Constant Blinding PRNG in Flash Player

3.1 Introduction

Constant blinding mechanism is adopted in JIT compilers to defend against JIT spray attacks. It aims at preventing the constant under attacker 's control appearing in code heap. For this purpose, constant blinding relies on a PRNG to generate a random number. The number serves as a secret cookie to scramble the value of the constant that will be emitted in memory. Therefore, the PRNG plays a significant role in constant blinding. If an attacker can recover the seed value of the PRNG under reasonable time cost, the scrambled value will be predictable and the attacker will gain the ability to put ROP gadgets into code heap via JIT-spray attacks at his own will.

Since the critical role of PRNG in modern security systems, it has been analysed in several previous works. A commonly used method in attacking a PRNG is to lower to entropy used in the PRNG. This can be done by exploiting the weakness in the external entropy source. Kim et al. [86] analysed the OpenSSL PRNG on the Android system. They demonstrated that the lack of entropy at the time of seed

initialization will make it vulnerable for attackers to predicate the state of PRNG and recover the secret key of SSL session. Similar work [85] was also done on Linux PRNG. It demonstrated that the low entropy provided by the PRNG will make the device vulnerable to IPv6 fragmentation attack and stack canary bypass. Constant blinding PRNG in Flash Player tries to design an efficient PRNG algorithm itself for providing secret cookie while reducing the performance overhead. Our research shows that the flawed design and ill-considered implementation still make attackers completely recover the seed status and launch JIT-spray attacks.

We analyse the design and implementation of the PRNG for constant blinding in Flash Player. We propose a novel attack to lower the entropy of the PRNG. Instead of analysing the external entropy source like the previous work [85, 86], we make cryptanalysis on the PRNG algorithm itself. Due to insufficient confusion and diffusion of the PRNG, we can reduce the entropy in the secret generated cookie to only 21-bit. We can recover the seed value in less than one second. Moreover, we propose another method to recover the seed value in the PRNG in Flash Player. We find that the seed value is stored in heap in the implementation of the PRNG and it is always located next to some constant values. We can use those constant values as signatures to locate the seed value in memory. The search for the seed value only requires $\mathcal{O}(1)$ time, which is fast and stable. Compared with previous work [89, 90] that only searches for some corner cases in constant blinding, we are the first to analyse the PRNG in constant blinding and predict the secret cookie by recovering seed value.

We further propose and implement a dynamic binary instrumentation framework ConBE (Constant Blinding Enhancement) based on PIN to mitigate our attacks on PRNG. We introduce extra entropy in the generation of pseudo number, making cryptanalysis on PRNG impractical. We also move the seed value to an isolated heap, making it hard for attackers to search in memory. Compared with the latest patch of Flash Player, we provide a stronger protection that an attacker cannot recover the

seed value even if the attacker gains the ability to arbitrarily read or write in memory. We evaluate our mitigation for Flash Player on Windows 7 platform and compare the performance overhead with the patched version of Flash Player.

In summary, we make the following three contributions:

- Analyse the design and implementation of PRNG for constant blinding in Flash Player. Propose two feasible methods to circumvent constant blinding based on cryptanalysis and information disclosure respectively.
- Present the proof-of-concept exploit based on existing vulnerability and evaluate its performance overhead.
- Propose a prototype mitigation against our attack based on dynamic binary instrumentation framework.

3.2 Attacking Model

In our attacking model, we assume that the attacker gains the following abilities.

- The target is under the protection of DEP [4] and ASLR [38], which is enabled at the hardware level and OS level.
- The target is vulnerable to memory corruption. Hence, the attacker can read arbitrary value in memory and modify value at given address.
- The target is assumed to be protected by code diversification techniques, such as G-free, that there exists no available gadget in static code or dynamically generated code.

We take Flash Player as our target with DEP and ASLR in place. DEP is a defence against shellcode injection attack. DEP prevents attackers from directly writing shellcode into executable code heap and hijacking the control flow to the shellcode.

ASLR is another defence that raises the bar of successful exploitation. Through randomizing the base address of loaded module and mapped memory, attackers have to exploit information exposure vulnerability to disclose critical information before hijacking control flow. We will discuss more academic work on protection for JIT code in Section 3.7.2.

The second one is a common practice for developing stable exploits today. Attackers usually exploit various kinds of vulnerabilities in target [111], including use-after-free, type confusion, etc. Recent exploits [8,9] on Flash Player show that ActionScript provides attackers chances to trigger vulnerability, leak critical information in memory. Therefore, ASLR is not a big concern in our attacking model.

3.3 Attacks on Constant Blinding In Flash Player

As described above, security of constant blinding heavily relies on the secret cookie generated. If the secret cookie is predictable for attackers, the defence against JIT spray attack will be weakened. In Section 3.3.1, we will discuss the implementation details of constant blinding PRNG in Flash Player. Next, we will demonstrate how we recover the seed value with two different methods in less than 1 second. The first method is to apply cryptanalysis on the hash function. We will show that the output of PRNG algorithm can be reversed at $\mathcal{O}(2^{21})$ time complexity in Section 3.3.2. The second method is to search the seed value directly. Different from searching the whole memory space to locate the seed value, we find the seed value in memory at $\mathcal{O}(1)$ time cost based on pointer redirection in Section 3.3.3.

3.3.1 Constant Blinding in Flash Player

The PRNG contains three components as shown in Listing 3.1: a seed initialization function ($RandomFastInit$), a hash function ($RandomPureHasher$) and a generator function ($GenerateRandomNumber$) to generate the final secret cookie for constant

```
void RandomFastInit(pTRandomFast pRandomFast) {
 pRandomFast->uValue = (int)(getPerformanceCounter());
 pRandomFast->uSequenceLength = 0x7fffffff;
 pRandomFast->uXorMask = 0x14000000;
}

int RandomPureHasher(int iSeed) {
 int iResult;
 iSeed = ((iSeed≪13)⊕iSeed)−(iSeed≫21);
 iResult = (iSeed∗(iSeed∗iSeed∗c3 +c2) + c1);
 iResult = iResult & kRandomPureMax;
 iResult = iResult+iSeed;
 iResult = ((iResult≪13)⊕iResult)−(iResult≫21);
 return iResult;
}

int GenerateRandomNumber(pTRandomFast pRandomFast) {
 if(pRandomFast->uValue == 0){
    RandomFastInit(pRandomFast);
 }
 long aNum = RandomFastNext(pRandomFast);
 aNum = RandomPureHasher(aNum * 71L);
 return aNum & kRandomPureMax;
}
```

Listing 3.1: Constant Number Generation Process

blinding. Another function (*RandomFastNext*) generates a number based on the seed value and updates its value, but it does not change the entropy in the seed value.

In initialization function, the seed value $uValue$ is initialized by QueryPerformanceCounter (Windows API). The hash function takes the seed value as input and generates a hash value. The new value will be ANDed with kRandomPureMax (`0x7fffffff`) in the generator function to produce the final cookie.

In hash function, the variables $c1, c2, c3$ are three constant numbers. This hash function adds no extra entropy into the generated number but aims to make it hard for attackers to reverse the seed value. Attacker's goal is to retrieve the seed value, predicate the secret cookie generated in next round and embed the desired value in the executable code heap. Though reversing the seed value via brute force seems feasible, it is impossible in practice because the default running timeout in Flash Player is 15 seconds. An attacker must recover the seed value in less than 15 seconds

while brute forcing requires a few minutes on average according to our test.

3.3.2 Attack Based On Cryptanalysis

Hash function is supposed to make attackers unable to reverse the seed value under reasonable time cost. However, we find that the hash function used for constant blinding in Flash Player is insufficient in confusion and diffusion, such that the attacker still can get the seed value in short time. The hash function in Listing 3.1 can be simplified into two functions. The first one is a bit manipulation function $f(n)$, and the other is a third-degree polynomial function $g(n)$. These two functions can be generalised as following equations:

$$f(n) = ((n \ll 13) \oplus n) - (n \gg 21) \tag{3.3.1}$$

$$g(n) = (c_3 * n^3 + c_2 * n + c_1)\&\texttt{0x7fffffff} + n \tag{3.3.2}$$

For the polynomial function, we can easily reverse the input value. The simplified algorithm is given in Algorithm 1. The reverse algorithm starts to scan the target value from the least significant bit (Line 7). Then, we apply a backtracking Algorithm 2 to reverse the input value bit by bit. We guess one bit and use the polynomial function to verify (Line 7 and 13) if the result matches the first i bits of target value.

For the bit manipulation function, we design Algorithm 3 to reduce the time complexity to $\mathcal{O}(2^{21})$, which is more efficient compared with $\mathcal{O}(2^{32})$ of brute-force method. To recover the seed value of bit manipulation function, we express the seed value in a 32-digit string as $a_1a_2a_3...a_{31}a_{32}$ and express target value as $t_1t_2t_3...t_{31}t_{32}$. We reorganize the bit manipulation function $f(n)$ and present the process of bit manipulation function in Figure 3.1.

According to the generalized equation, the seed value is divided into three parts. The higher part is $a_{01}...a_{11}$, the middle part is $a_{12}...a_{21}$ and the lower part is $a_{22}...a_{32}$. Since target value is known to us, attack on PRNG can be divided into two steps. We can recover the value of higher part and lower part first and then reverse the

Algorithm 1 Reverse Polynomial Function

1: **Function name:** Reverse
2: **Input:** V: the value to reverse
3: **Output:** S: the set of candidate value
4: $S = \emptyset$
5: target_bit = 0
6: curSolution = 0
7: reverseBit(V, target_bit, curSolution, S)
8: **return** S

$$
\begin{array}{ll}
 & a_{01}a_{02}a_{03}a_{04}a_{05}a_{06}a_{07}a_{08}a_{09}a_{10}a_{11}a_{12}a_{13}a_{14}a_{15}a_{16}a_{17}a_{18}a_{19}a_{20}a_{21}a_{22}a_{23}a_{24}a_{25}a_{26}a_{27}a_{28}a_{29}a_{30}a_{31}a_{32} \\
\oplus & a_{14}a_{15}a_{16}a_{17}a_{18}a_{19}a_{20}a_{21}a_{22}a_{23}a_{24}a_{25}a_{26}a_{27}a_{28}a_{29}a_{30}a_{31}a_{32}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
= & t_{01}t_{02}t_{03}t_{04}t_{05}t_{06}t_{07}t_{08}t_{09}t_{10}t_{11}t_{12}t_{13}t_{14}t_{15}t_{16}t_{17}t_{18}t_{19}t_{20}t_{21}t_{22}t_{23}t_{24}\ t_{25}t_{26}t_{27}t_{28}t_{29}t_{30}t_{31}t_{32} \\
+ & 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ a_{01}a_{02}a_{03}a_{04}a_{05}a_{06}a_{07}a_{08}a_{09}a_{10}a_{11}
\end{array}
$$

Figure 3.1: Reorganized hash function bit by bit

possible value of middle part. For higher part and lower part, we iterate over possible values of the lower part (Line 5) and reverse the value of higher part through value in $t_{22}...t_{32}$ (Line 6–11). For middle part, we iterate over possible values of the middle part (Line 12) and use the known value of $t_1..t_{21}$ to verify if the value is valid (Line 14–17). Therefore, the time complexity of the whole process will be $\mathcal{O}(2^{11}) * \mathcal{O}(2^{10}) = \mathcal{O}(2^{21})$.

3.3.3 Attack Based on Memory Disclosure

Besides the attack based on cryptanalysis, the implementation of the PRNG ignores the possibility that the attacker can locate the seed value in memory through searching memory space via info disclosure vulnerability.

To find the seed value quickly and stably, we have to solve two challenges during our search in memory. The first challenge is how to assure that the accessed value is the seed value we are searching for. The second challenge is to get the memory address of the seed value in the heap.

For the first challenge, the solution is straightforward. In initialization function of

Algorithm 2 Backtrack Algorithm to Reverse Polynomial Function bit by bit

```
1: Function name: reverseBit
2: Input: V: the value to reverse, i: the i_th bit to check, sol: current solution, S: the set to store
   candidate value
3: if i == 32 then
4:     S.add(sol);
5:     return
6: end if
7: if verify_ith_bit(V, i, 0) then
8:     sol = sol + 0≪i
9:     i = i + 1
10:    reverseBit(V, i, S, sol)
11:    i = i - 1
12:    sol = sol - 0≪i
13: end if
14: if verify_ith_bit(V, i, 1) then
15:    sol = sol + 1≪i
16:    i = i + 1
17:    reverseBit(V, i, S, sol)
18:    i = i - 1
19:    sol = sol - 1≪i
20: end if
21: return
```

the PRNG, the seed value is initialized together with sequenceLength and XorMask. However, the values of those two variables are constant numbers and these two values are located in memory adjacent to the seed value. These two variables can be used as signature values for attackers to locate the seed value.

For the second challenge, the difficulty comes from the fact that the seed value and the victim $Vector$ object are located in two different heaps. Usually, a successful exploit on Flash Player relies on corrupting the metadata *length* in a $Vector$ objectt, which enables attacker to arbitrarily read memory out of the bound of original buffer. Attempt to read value at given address, which is not readable or unmapped, will cause

Algorithm 3 Reverse algorithm for the polynomial function

```
1: Function name: reverseHash
2: Input: V: the value to reverse
3: Output: S: the set of candidate value
4: S = ∅
5: for lowPart ∈ [0, 0x7ff] do
6:     tmpPart = V & 0x7ff
7:     if tmpPart>lowPart then
8:         highPart = (1≪11) + lowPart - tmpPart
9:     else if tmpPart<=lowPart then
10:        highPart = lowPart - tmpPart
11:    end if
12:    for midPart ∈ [0, 0x3ff] do
13:        value = highPart≪21 + midPart≪11 + lowPart
14:        t = (value≪13)⊕value - (value≫21)
15:        if t == V then
16:            S.add(value);
17:        end if
18:    end for
19: end for
```

access violation. To avoid unnecessary crash in our exploit, we have to figure out a reliable and quick way to locate the heap where the seed value is located. Our solution is to create an anchor object and save the reference to anchor object in the same heap as victim vector. The anchor object contains a reference to another object which is located in the same heap as the seed value. Through chains of pointer indirection, we can locate the heap storing the seed value. Since size of the heap is a constant number, time complexity to search for the seed value is $\mathcal{O}(1)$.

3.3.4 Full Exploit Generation

In our exploit, we call a function as constant releasing function if its generated code contains a blinded constant. In Listing 3.2, we list two simple constant releasing

```
function fun1() { a = 0x41414141; }
function fun2() { c = 0x43434343; }
fun1();
fun2();
```

Listing 3.2: Benign code for demonstrate how blinded constant are generated

functions and invoke them one after another in script. From the view of constant blinding, the steps to emit code can be separated into a few steps.

1. Generate JIT code for fun1
    - Call *GenerateRandomNumber* and get Key1
    - Retrieve `0x41414141` from symbol table
    - Generate JIT code: mov ecx, `0x41414141` $\oplus$ Key1
    - Generate JIT code: xor ecx, Key1
2. Generate JIT code for fun2
    - Call *GenerateRandomNumber* and get Key2
    - Retrieve `0x43434343` from symbol table
    - Generate JIT code: mov ecx, `0x43434343` $\oplus$ Key2
    - Generate JIT code: xor ecx, Key2

As discussed in Section 2.2.2, constants used in the script are stored in a symbol table. Since the constant values in this table are not blinded. We can locate the address of the table by inserting some magic constants in script. If we can predicate the secret cookie in next round, we can modify the value in the table and embed desired 4-byte long value in code heap in the end.

In our final exploit, we insert malicious code before generating JIT code of $fun3$. In malicious code, we trigger the vulnerability, search in memory and recover the seed value via given methods. To embed blinded constants in memory, we need

```
function fun1() { a = 0x41414141; }
function fun2() { c = 0x42424242; }
function fun3() { c = 0x43434343; }
fun1();
fun2();
evilCode();
fun3();
```

Listing 3.3: Malicious code to retrieve seed value and embed desired value in code heap

to call multiple constant releasing functions before running our malicious code in Listing 3.3. Since the hash function is a many-to-one function, we may get multiple possible seed values if we are given only one secret cookie in the process of reversing the hash function. Leaking multiple secret cookies can help us find the real seed value. The necessary steps can be generalized:

1. Generate JIT code for fun1
    - Call *GenerateRandomNumber* and get Key1
    - Retrieve `0x41414141` from symbol table
    - Emit binary code: mov ecx, `0x41414141` $\oplus$ Key1
    - Emit binary code: xor ecx, Key1
2. Generate JIT code for fun2
    - Call *GenerateRandomNumber* and get Key2
    - Retrieve `0x42424242` from symbol table
    - Emit binary code: mov ecx, `0x42424242` $\oplus$ Key2
    - Emit binary code: xor ecx, Key2
3. evilCode: search and modify memory under attacker's control
    - Search memory to find Key1 and Key2

- Recover seed value, and predict the value of Key3
- Locate `0x43434343` in symbol table
- Modify `0x43434343` to be (TargetValue $\oplus$ Key3)

4. Generate JIT code for fun3
   - Call *GenerateRandomNumber* and get Key3
   - Retrieve (TargetValue $\oplus$ Key3) from symbol table
   - Emit binary code: mov ecx, TargetValue (TargetValue $\oplus$ Key3 $\oplus$ Key3)
   - Emit binary code: xor ecx, Key3

3.4 Design and Implementation of ConBE

To mitigate our attacks, we propose Constant Blinding Enhancement (ConBE) based on dynamic binary instrumentation as a prototype to demonstrate our mitigation strategy. Adobe has patched this vulnerabilities in the latest version of Flash Player (25.0.0.127). We build this tool to provide stronger protection for all versions of Flash Player. In this section, we will discuss our mitigation strategy and implementation in detail.

3.4.1 Mitigation principles

To mitigate the attack based on cryptanalysis, we make it harder for attackers to reverse the hash function. At present, we rely on well-documented hash function to achieve this. To protect seed value from information disclosure, our solution is to separate the seed value from those signature values and store the seed value in an isolated heap. Similar to partitioned heap, we invoke $VirtualAlloc$ function to create a new data heap to store the seed value. It means that except for one global pointer referencing the seed value in the heap there will be no other data pointer in memory

referencing any values in the data heap. At present, our solution relies on ASLR provided by OS to randomize the heap location.

3.4.2 Implementation details

We build ConBE based on PIN tool [88], a dynamic binary instrumentation framework. As a prototype, we use ConBE to test the effectiveness of our mitigation strategy and provide protection for older versions of Flash Player.

To mitigate cryptanalysis, we insert an MD5 hash function at the end of identified hash function. In particular, we take the value in *eax* as input value and calculate its MD5 hash value. The output of the MD5 hash function is a 128-bit value. We pick the least significant 32 bits as its return value.

To thwart information disclosure attack, we have to achieve two goals. The first one is to separate the seed value from those signature values. The second one is to hide the seed value in memory. Different from X64 platform, X86 system does not have a free segment register to save the value in an isolated memory segment. To implement both goals, our solution is to newly allocate a heap and save the seed value in the heap. We rely on ASLR provided by system to randomize address of the memory. Under our protection, an attacker has to search through the whole memory. It takes too much time and exceeds the default timeout of ActionScript.

3.4.3 Advantages of ConBE

The latest PRNG in Flash Player relies on Windows API *CryptGenRandom* to generate the random number. To be more specific, the PRNG maintains a secret buffer of 256 bytes long and invokes the API once to fill the secret buffer with random values. As a result, 256/4=64 secret cookies will be available and stored in the secret buffer for future use. For each user-defined function, JIT compiler traverses the secret buffer and pick an unused 4-byte value as secret cookie.

Once all the 64 secret cookies have been used once, *CryptGenRandom* will be called again to generate a new secret buffer. Partitioned heap in Flash Player is responsible for assuring that the secret buffer will not be located in the same heap as victim object is located. Allocating a 256-byte long buffer for storing secret cookie is a trade-off between performance and security. In Section 3.5.2, we demonstrate the performance overhead of putting the seed value in a separate heap is pretty high. Generating 64 secret cookies in a row rather than invoke *CryptGenRandom* for each user-defined function can improve performance overhead. Since the seed values are still stored in data segment, dedicated attackers may still locate the seed value with generic information disclosure technique. On the contrast, ConBE could be customized on X64 system to store the only pointer that references the seed value in extra segment rather than data segment to make it harder for attackers to search in memory space.

3.5 Evaluation

In Section 3.5.1, we will give a full exploit to recover the seed value of PRNG based on one existing CVE in Flash Player. Then we evaluate the execution time of our attack. Then we evaluate the performance overhead of our proposed prototype mitigation. In Section 3.5.2, we evaluate the performance overhead of ConBE and compared that with the patched Flash Player. The performance evaluation is tested in a Windows 7 running on Intel Xeon E5-2630 processor with 4GB RAM.

3.5.1 Evaluation on Full Exploit

To show the effectiveness of our attack, we present a proof-of-concept exploit based on CVE-2015-5119 [8], a use-after-free vulnerability in Flash Player 18.0.0.206 [1].

In the exploit of CVE-2015-5119, an attacker can corrupt the metadata length of

[1]The weak PRNG exists from version 2 to version 24.0.0.221 of Flash Player.

a *Vector* object and utilizes the extended vector to gain arbitrary read/write ability in memory. As discussed in Section 3.3.3, we use a *ByteArray* object as an anchor to leak the address of the heap where the seed value is stored. More specifically, there exists a reference to our target heap at the offset 0x30 in *ByteArray* object. We first discover the address of a *ByteArray* object, read the reference to locate the target heap and then search through memory for the signature value to locate the seed value.

Table 3.1 demonstrates the time needed to retrieve the secret value compared to a normal execution. We use internal function of ActionScript *Date.getMiliseconds* and *Date.getSeconds* to get the execution time. In both normal execution and poc exploit, we pick the time when we corrupt the metadata of victim vector as starting time. For normal execution, we take the time when all blinded constants are released as ending time. For poc exploit, we take the time when the exploit finds all necessary ROP gadgets as ending time. From the table, we can see that the Information Disclosure method and Cryptanalysis method induce 0.033s and 0.266s overhead respectively comparing to the normal execution. From the view of an exploit, such performance overhead is tolerable and will not influence much on the exploit performance.

3.5.2 Evaluation on ConBE

We have successfully applied our tool in Flash Player 18.0.0.203. To evaluate the effectiveness of ConBE, we have to answer two questions: (Q1) What is the performance overhead induced by ConBE? (Q2) What is the difference on performance between ConBE and the officially patched version of Flash Player?

To answer Q1, we have to compare the performance overhead between ConBE patched one and the original unpatched one. To answer Q2, we need to compare the performance overhead between ConBE patched one and the latest officially patched one. For the ConBE patched one and the original unpatched one, we calculate the time between fetching seed value from memory and generating final secret cookie. Be-

Table 3.1: Time required to retrieve the secret value

Tested Mode	Required Time
Normal	1.732
Information Disclosure	1.765
Cryptanalysis	1.998

Table 3.2: CPU cycles required to generate secret number

Flash Player Version	CPU cycles
18.0.0.203	1113660
18.0.0.203 (ConBE)	4151108
25.0.0.127	3155080

cause the patched version of Flash Player uses the Windows API *CryptGenRandom* to generate secret value, we calculate the execution time of the API call.

Since internal function *Date.getMiliseconds* of ActionScript only provides millisecond resolution, we can tell no difference in the time latency using those internal functions. Instead, we use PIN to insert instructions to log the CPU cycles used to generate the secret value. To be more specific, we instrument *rdtsc* instruction to log the CPU cycles.

Table 3.2 demonstrates the CPU cycles to generate a secret number. Time to generate the secret cookie in officially patched one is about 2.8 times that in the original one. On the contrary, ConBE is about 3.7 times that of the original one. We think the time delay is resulted from the following two reasons. First, we use an MD5 hash function and allocate a separate heap to save seed value in our implementation. But for the patched version, it relies on existing partitioned heap to store the generated secret cookie. The secret cookie is stored in a heap together with some security-unrelated data. This process in ConBE is more complicated than the patched one. Second, our security enforcement is built on PIN, a dynamic instrumentation

framework. Similar works [70, 75, 121] built on PIN have shown that this framework usually induced a high performance overhead.

3.6 Discussion

In this section, we are going to discuss some questions that may arise in this paper. In Section 3.6.1, we are going to discuss why we take study on the constant blinding PRNG in Flash Player and how serious it could be with a weak PRNG in modern system. In Section 3.6.2, we are going to discuss some more details of the latest patch and explain what motivates us to implement ConBE based on PIN.

3.6.1 Implication of PRNG Bypass

Nowadays, many modern mitigations heavily rely on the randomness provided by the PRNG. The low-entropy PRNG will weaken the basis those mitigations are built upon. Kim et al. [86] analyse the OpenSSL PRNG on the Android system. They demonstrated that the lack of entropy at the time of seed initialization will make it easier for attackers to predicate the state of PRNG and recover the secret key of SSL session. Similar work [85] is also done on Linux PRNG. It demonstrates that the low randomness provided by the PRNG on embedded system will make the device vulnerable to IPv6 fragmentation attack and stack canary bypass.

In this paper, Flash Player tries to design an efficient PRNG itself for providing secret cookie used in constant blinding while reducing the performance overhead. Our research shows that the flawed design and ill-considered implementation still make attackers completely recover the seed status and launch JIT-spray attack.

3.6.2 Latest Patch of Flash Player

As discussed in Section 3.5.2, the latest PRNG in Flash Player relies on Windows API *CryptGenRandom* to generate the random number. To be more specific, the PRNG

maintains a secret buffer of 256 bytes long and invokes the API once to fill the secret buffer with random values. As a result, 256/4=64 secret cookies will be generated and stored in the secret buffer for future use. For each user-defined function, JIT compiler traverse the secret buffer and pick an unused 4-byte value as secret cookie.

Once all the 64 secret cookies have been used once, *CryptGenRandom* will be called again to generate a new secret buffer. Partitioned heap in Flash Player is responsible for assuring that the secret buffer will not be located in the same heap as victim object. Allocating a 256-byte long buffer for storing secret cookie is a trade-off between performance and security. In Section 3.5.2, we demonstrate the performance overhead of putting the seed value in a separate heap is pretty high. Generating 64 secret cookies in a row rather than invoke *CryptGenRandom* for each user-defined function can improve performance overhead. Since the seed values are still stored in memory, dedicated attackers may still locate the seed value with generic information disclosure technique. On the contrast, ConBE could be customized to store the only pointer that references the seed value in extra segment rather than data segment to make it harder for attackers to search in memory space.

3.7 Related Work

3.7.1 Other Techniques Bypassing Constant Blinding

Athanasakis et al. [44] focus on the fact that constant blinding does not blind constant whose length is less than 2 bytes. Therefore the attacker is still able to construct enough useful ROP gadgets for a successful exploit under 64-bit system. However, our work demonstrates that the attacker is able to allocate 4-byte long ROP gadget in memory, which provides more flexibility and availability of ROP gadgets. Moreover, our attack works on both 32-bit and 64-bit system.

In the work of Maisuradze et al. [89], the authors proposed that the relative jump offset can also be used to allocate desired 2-byte or 3-byte ROP gadget in memory.

However, attacker needs to validate the emitted gadget before chaining them as ROP gadget because of NOP sledding adopted by most JIT rendering engine. On the contrary, our attack on constant blinding does not require the process of validation and ensures that the desired ROP gadgets will always be emitted in memory.

Dachshund [90] finds some corner cases, which was not covered by constant blinding, to emit desired target value in JIT code. In Chakra, the script engine of Mocrosoft Edge, it is discovered that the constant number in function argument is not blinded by secret value. In Chrome, There also exist a few corner cases where constant blinding does not take place. To the best of our knowledge, we are the first to analyse the PRNG in constant blinding and predict the secret cookie for bypassing constant blinding. Moreover, flash player is still supported on all main stream browsers nowadays, our work provides a more general and stable attacking vector to emit executable gadget in memory.

3.7.2 Protection on JIT Code

INSert [126], RIM [128] and JITSafe [57] all propose techniques to defend against JIT-Spray attack. To prevent immediate value from being emitted in code heap, they all rely on a secret value to obfuscate the immediate value. The security of these mitigation is based on the assumption that the secret value is not predictable to attackers. However, our work shows that the assumption is not guaranteed in its implemenation of Flash Player. Rock-JIT [97] implements a CFI enforcement on JIT compilation. For JIT compiler, Rock-JIT builds the Control Flow Graph (CFG) of static code and enforces a fine-grained CFI. Meanwhile, Rock-JIT adopts a coarse-grained CFI for JITted code. Rock-JIT maintains a set to remember all starting instruction of all JITted code and insert check to verify the validity of target before indirect jump. Rock-JIT is built on source code of Chrome V8 script engine and is hard to evaluate its effectiveness in Flash Player, which does not make its source code public. DSCG [115] aims at preventing possible attacks that exploit the code

cache in Chrome V8 engine, whose permission is writeable and executable at the same time. On the contrary, Flash Player has taken such attack into consideration. The permission of code heap is writeable and readable at the time of emitting JIT code, while the permission of code heap is executable and readable at the time of executing JIT code. The short attacking window makes it less likely for attackers to launch such exploit. However, our work shows that even if the code heap is enforced with W$\oplus$E, an attacker is still able to emit some desired gadgets in code heap.

Chapter 4

Discovering Non-Blinded Constant in Browser JIT Engine

4.1 Introduction

In recent works, much focus has been put on discovering cases where constant blinding does not cover. In the work of Athanasakis et al. [44], 2-byte values in JavaScript, which are not protected by constant blinding, are demonstrated sufficient to provide code gadgets for a complete ROP chain. Maisuradze et al. [89] proposed the concept of implicit constant and demonstrated relative jump/call offset will be used as non-blinded constant. Though such implicit constant is not in the protection scope of constant blinding, nop sledding will insert some meaningless instructions in the JIT code and the relative call/jump offset will be not the same as expected at times. Dachshund [90] is proposed as a fuzzing framework to search for corner cases of explicit number where constant blinding does not take place. Dachshund inserts specific number, e.g. `0xc35941` ($pop\ r9; ret$) and `0xc35841` ($pop\ r8; ret$), in script code and tests if those numbers appear in the generated code. Dachshund aims at discovering potential explicit constant numbers that appear in JIT code.

The insight of our approach is that the implicit size or implicit offset in script may

also lead to emitting non-blinded constants in JIT code. The challenge for us is how to generate the test script that reflects such information. In this chapter, our solution is to find the repeated elements that can be displayed in the structure of a script. The element here not only refers to the primitive element in a programming language, like variable or object, but also refers to the complex element, such as statement or expression.

To this end, we choose grammar-based fuzzer to generate test scripts in order to search for non-blinded constants. Grammar-based fuzzer generates the fuzzing input from a given grammar. Compared with dumb fuzzer that randomly mutates the bytes in seed file, the input file generated by grammar-based fuzzer is usually well-formed and less likely to cause crash. We take a deep look into the grammars of JavaScript and find the target types of production rule that can guide us in generating the input script for fuzzing test. Based on the grammar collected from our analysis, we build Blockade, a grammar-based fuzzing framework, to detect the non-blinded constant in JIT code. In this framework, the fuzzer takes the value that represents target ROP gadget as input (e.g. `0xc35941` for $pop\ r9; ret$) and generates the test file for fuzzing. We later feed the file to target application and search for the target value in memory.

We run our tool on Microsoft Edge and Google Chrome to test the effectiveness of our framework. We find that except for large explicit number, attackers can take advantage of various kinds of structural information in script language to emit desired values in the JIT code. Besides the relative jump/call offset that has been discussed in previous work, our result shows that object field, global variable and even number of statements in script can be used as implicit constant to release potential ROP gadget. The relative jump/call offset is one of the cases that are derived from our collected grammar. As for explicit constant, our discovery shows that the constant that appears in JIT code does not have to be identical to the number inserted in script. Such a discovery may assist us find more non-blinded explicit constant in future.

In this chapter, we make the following two contributions:

- We propose and implement Blockade, a grammar-based fuzzing framework, to detect non-blinded constant in JIT code. In this fuzzing framework, we specify the necessary grammar as a guide for generating input script and propose algorithms for generating input files more efficiently.
- We test our tool on Microsoft Edge and Google Chrome. Including the cases that have been discovered in previous work, we find more cases where non-blinded constant numbers are JITted into code heap. Our work demonstrates the incompleteness of constant blinding against JIT spray attack in current script rendering engine of mainstream browsers.

4.2 Overview of Blockade

Before introducing our approach for discovering non-blinded constants in JIT code, we have to clarify the threat model and attacking goal in our attack. Then we will introduce the insight of our approach for discovering non-blinded constants in JIT code and give an overview of the components in Blockade.

4.2.1 Threat Model

To evaluate the effectiveness of our proposed fuzzing framework, we assume that the target binary follows the conditions below:

C1 The target is under the protection of W⊕X provided by operating system (DEP [5] on Windows, or PaX [31] on Linux).

C2 The target is vulnerable to information leakage. Attacker is able to arbitrarily read value in a given address.

C3 The target is assumed to be protected by code diversification techniques, such as G-free, that there exists no available gadget in static code or dynamically generated code. But the target provides attackers a scripting environment to generate JIT code on the fly.

C4 We assume that constant blinding and NOP sledding are employed as JIT defense against JIT spray attack in the target binary.

With **C1**, attackers cannot inject shellcode directly into memory for exploit. As discussed in work of Serna et al. [111], information disclosure may result from various kinds of vulnerabilities such as heap overflow, type confusion or use-after-free. Info leak will become a common technique in software exploitation [8,25]. So we think **C2** is reasonable and practical in real world. Moreover, we find that the JIT code heap in Chrome is with RWX permission, which breaks **C1**. Therefore, we do not assume that the attacker gains the ability to arbitrarily modify data in memory under our attacking model. In **C3**, we only consider the ROP gadget embedded in JIT code and JIT compiler provides a chance for attackers to emit code. As for **C4**, we do not take the sandbox or CFI into consideration. The main focus of this paper is to discover the non-blinded constant that evades constant blinding. Discussion on how to circumvent sandbox and CFI is out of the scope of this paper.

4.2.2 Attacking Goal

In this paper, our attacking goal is to generate proper gadgets for ROP chain construction. As depicted in Figure 2.1, if an attacker is able to load value from stack into argument register $(r8, r9)$, the attacker will control the argument value for most of critical system calls including $WinExec$. Therefore, in this paper we aim at finding the constant numbers, `0xc35941` $(pop\ r9; ret)$ and `0xc35841`$(pop\ r8; ret)$, which are emitted in the JIT code. The constant value is the input value to our fuzzer as a parameter. We only need to change the constant number if we want to search for

other non-blinded constants in JIT code. Since argument values are passed through stack, attackers do not need to find gadgets to load value into argument register on X86 system. Therefore we only focus on X86-64 system in this paper.

Compared with previous work, we decide that our tool has to achieve three requirements.

R1 The identified non-blinded constants are at least 3-byte long.

R2 Besides relative jump/call offset, the tool must discover more implicit non-blinded constants in JIT code that have not been discovered before.

R3 The tool is able to discover the cases where explicit numbers are not blinded.

In this paper, we take Microsoft Edge and Google Chrome as our target. Constant blinding and nop sledding are both employed in the two browsers. We pick **R1** because 2-byte or 1-byte constant values are not protected by constant blinding in both browsers for improving performance. We have to show that our tool is able to find non-blinded constants, which should have been blinded in Edge and Chrome. Moreover, a 3-byte value in JIT code will provide more convenience in ROP chain construction. In previous work, only call/jump relative offset is proposed one type of implicit constants that evade constant blinding. But it is still unknown if there exists other cases of implicit number. Therefore, we take **R2** to demonstrate that our tool is powerful enough to search for more types of implicit constant in JIT code. Since Dachshund has given a fuzzing framework for searching explicit number that constant blinding fails to cover, we pick **R3** to prove that our tool is also well designed to search for explicit number in JIT code.

4.2.3 Insight of The Approach

In the work of Xuanwu Lab [2], researchers propose that array offset can be used to provide non-blinded constant numbers. In Listing 4.1, the index into array is given

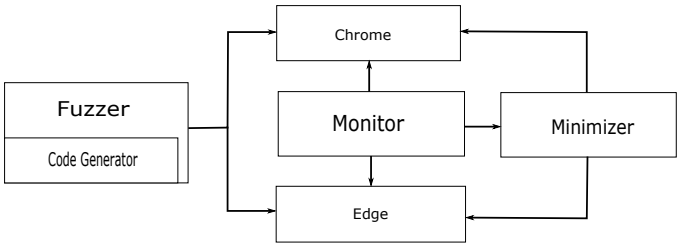


Figure 4.1: Overview of Blockade

```
script code:
var ar = new Uint16Array(0x10000);
ar[0x9090/2] = 0x9090;

JIT Code:
mov    word ptr [esi+9090h],9090h

Assembly code of JIT code:
66c786909000009090
```

Listing 4.1: Array Offset in Chakra Engine

as 0x00009090, a 2-byte value under control and additional 2-byte 0x00. Such a non-blinded number is sufficient for ROP construction.

In this case, it is the array offset that provides non-blinded constant. The root cause is that the constant blinding in JIT-compiler may omit the possibility of explicit offset or even implicit offset that exist in script language. The explicit offset can be array offset in the sample code. The implicit offset can be the call/jump relative offset as demonstrated in previous work. Therefore, we need a method to generate script that can reflect such explicit or implicit offset in script code. Inspired by the sample, we think the prerequisite for such information is the repeated elements displayed in the script. In another word, any element that can be repeatedly allocated in script may bring implicit or explicit offset in the structure of generated code. Driven by this, we decide to use a grammar based approach to find out cases of non-blinded constants in JIT code as many as possible. To address this, we are faced with two problems to solve:

Q1 What kind of grammar do we need to collect from JavaScript?

Q2 How can we generate the script code based on our picked grammar more efficiently?

We will demonstrate how we solve these two problems in Section 4.3.1 and 4.3.2.

4.2.4 Components in Blockade

We propose a grammar based fuzzing framework Blockade to detect potential non-blinded constant in JIT code. We choose to use a grammar based fuzzer for two reasons. First, grammar based fuzzer is perfectly matched with our purpose to discover potential non-blinded constants in JIT code with collected grammars. Secondly, we assume combination of different elements in script may generate unexpected JIT code in our test. Our test result also proves that such assumption is correct.

Blockade is consisted of three components: Fuzzer, Monitor and Minimizer. In Figure 4.1, we give an overview of Blockade. Fuzzer is for producing a piece of JavaScript code based on collected grammars, generate the input file and load the file into the target application. Monitor will search through the memory of script rendering process for target value and dump the file if target value exists in the JIT code of input script. Upon the dumped file from monitor, minimizer will remove the redundant lines in file while assuring the target value still exists in JIT code. In the following of the paper, we will introduce the design of fuzzing component in Section 4.3 and show how monitor and minimizer are expected to work in Section 4.4.

4.3 Fuzzing Approach

In this section, we are going to demonstrate how the fuzzing component in Blockade works. In Section 4.3.1, we will generalize the necessary grammars that assist us in generating input file. In Section 4.3.2, we demonstrate how we generate the test script

with given grammar. In Section 4.3.3, we will show how we generate the final input file for fuzzing test.

4.3.1 Context Free Grammar

To give a basic idea of our approach, we first give a Context Free Grammar (CFG) about array declaration in JavaScript. Usually, an array is defined as $\{1, 2, 3, "A", "1234"\}$. The CFG of array variable declaration is generalized below.

$$ArrayLiteral \rightarrow \{\} \mid \{ElementList\}$$
$$ElementList \rightarrow LiteralElement \mid ElementList\,,\, LiteralElement$$

In the sample code given in Listing 4.1, an array of size 0x10000 is created and the element at array offset 0x4848 is accessed. If we want to search for the non-blinded constants which are related with array offset, we must generate an array of large size first. To achieve this, we need the grammar that contains a recursive production rule. We can recursively add new element in array declaration as many times as we can. After having generated an array of large size, we attempt to access elements in the array and seach for non-blinded constants in JIT code. With this in mind, our target grammar contains at least one recursive production rule, which is of the form $L \rightarrow a|aL$. To search for all such grammars in JavaScript, we examine the grammar definition of ECMAScript 7.0 [7] and JavaScript 2.0 [23]. We have listed part of the identified grammars below. Besides the additive and multiplicative operation listed here, other operators, such as bitwise operator, logical operator, relational operator and etc, are also included in Blockade.

ArrayLiteral

$ElementList \rightarrow LiteralElement \mid ElementList\,,\, LiteralElement$

ObjectLiteral

$FieldList \rightarrow LiteralField \mid FieldList, LiteralField$

VariableLiteral

$VariableList \rightarrow VariableDeclaration$

$|VariableList\,,VariableDeclaration$

Arguments

$ArgumentList \rightarrow AssignmentExpression$

$|ArgumentList\,, AssignmentExpression$

Operator

$MultiplicativeExpr \rightarrow UnaryExpression$

$|MultiplicativeExpr * UnaryExpression$

$|MultiplicativeExpr/UnaryExpression$

$|MultiplicativeExpr\%UnaryExpression$

IFStatement

$IfStatement \rightarrow if\ (expression)\ Statements$

$|\ if(expression)\ Statements\ else\ Statements$

$|\ if(expression)\ Statements\ else\ IfStatements$

Statement

$Statements \rightarrow Statement\,|\,Statements\,,Statement$

Furthermore, we find that the script body itself can also be generalized as a recursive production rule as shown below. In our evaluation on Edge browser, the size of code heap is limited such that the a single script block (<script> codes </script>) cannot contain too many lines of code. Through this rule, we can separate the generated input code into different script blocks.

$MultiScript \rightarrow Script\ |\ MultiScript$

$Script \rightarrow < script > Statements < /script >$

4.3.2 Code Generator

In this section, we are going to demonstrate how we generate the input script based on collected grammars more efficiently. In another word, we have to decide times of recursion for a given production rule. Dachshund tries to only insert a specific number, e.g. `0xc35141` or `0xc35841`, in script to test if the explicit numbers are blinded or not. However, setting the times of recursion to a specific number may omit other potential numbers. Moreover, such a method will significantly increase the size the input file and make loading time of input script unacceptable in our fuzzing framework. However, randomly choosing a large number in a specific range (e.g. $[2^{17}, 2^{32}]$) with brute force is not efficient.

From the work in the past, we infer that the implicit number is linearly related with the times of recursion. We design a naive algorithm to generate a number, whose value floats near the target value. This number decides the times of recursion. In our experiment, we set target value to be `0xc35941` or `0xc35841`. Moreover, we also use this method to generate constant number (e.g. loop count and argument value) in input script at a certain probability.

In Algorithm 4, we give how we create the script code based on the grammar $Statements \rightarrow Statement|Statements, Statement$. After deciding the times of recursion, we generate a statement and concatenate the statement to the return string. The core idea is to randomly pick an element from the pool of available elements generated by our fuzzer and repeatedly add the element. Such a design is made based on a trade-off between increasing efficiency for input file generation and adding randomness for script fuzzing. We also apply same algorithm for other grammars identified in Section 4.3.1.

According to the given algorithm, the generation process will not stop unless $rnd(prob)$ return 0. Since $prob$ will decide how many different statements we will use in generated code, we set it to be 5 in our implementation of the tool at present. We assume the combination of different sequence of statements will add more variation in

Algorithm 4 Generation Algorithm for *GenStatements*

```
1: Input: targetValue
2: Output: sequence of JavaScript codes
3: code = "";
4: while !rnd(prob) do
5:     time = GenNumber();
6:     count= 0;
7:     stmt = genStatement();
8:     while count < time do
9:         code = code + stmt;
10:        count = count + 1;
11:    end while
12: end while
13: return code;
```

the JIT code after JIT compilation. Our result also proves such assumption is true in the case of number of statement. we will show how different combination of statements will influence the non-blinded constant in JIT code. Function *genStatement* is a basic generating function that generates statement from the existing grammar of JavaScript. Similarly, we also implement $genVariable$, $genExpr$, $genField$, etc.

4.3.3 Fuzzer

The fuzzer is responsible for generating the input file and loading the file into target application. The core part of fuzzer is a grammar based code generator. Based on recursive production rule in Section 4.3.1 and generation policy in Section 4.3.2, the code generator will generate a piece of JavaScript code. Then we will add some statements with magic values before the generated script code. These magic values will be later checked by Monitor to verify if JIT code was successfully generated. We emphasize that the magic value used in Dachshund is different from the magic value in our code generator. In Dachshund, a magic value is inserted in input script for discovering explicit constant that evades constant blinding. In our tool, a magic value

```
<html>
<script>
var ga = 0;
ga = ga + 0xdead;
</script>
<script>
/* Generate script code */
</script>
<script>
for(var i=0; i<LOOP_COUNT; i++)
{
        ga = ga + 0xbeef.
        /* Generate script code */
}
</script>
</html>
```

Listing 4.2: JavaScript Template for fuzzing

is only used for judging whether JIT code is successfully generated or not.

In the end, the fuzzer will insert the code into a JavaScript template in Listing 4.2 and feed the file to target application. In the template, expression `a=a+0xdead` and `a=a+0xbeef` are both statements with a magic value. We pick two 2-byte constant values as magic values, because 2-byte value will not be blinded in both Chrome and Edge.

An important difference of JIT compiler between Chrome and Edge is the timing to compile script into native JIT code. In Chrome, the JIT compiler will compile the script into JIT code at the first time running on it. In Edge, the script code will not be compiled into JIT code immediately. Initially, Edge will rely on an interpreter to run the script code. Only after the script code has been executed several times, the script code will be compiled by the JIT compiler in Edge. Therefore, in the template script for Edge, we insert a for loop to repeatedly run on generated code to assure that the JIT code of input script will be generated. In our implementation of the tool, we set $LOOP\_COUNT$ to 200 in our test on Edge and 1 in our test on Chrome.

4.4 Monitor and Minimizer

4.4.1 Design of Monitor

Monitor is expected to detect the target value in the script rendering process. More particularly, monitor will attach itself to a running process of Edge or Chrome and search for the target value in memory of the process. On X86-64 platform, the address space is too large to search for the magic value in memory with brute force. Therefore, we split the searching process into three steps: (1) Identify potential code heap in script rendering process; (2) Search for the magic value that is embedded in the input script; (3) Search for target value in the identified heap.

In Chrome, the JIT code is expected to lay in a heap with RWX permission. On the contrary, JIT code heap in Edge is with RX permission. Searching through the heap with RWX (in Chrome) or RX (in Edge) permission only will significantly reduce the searching time for target value.

There are various reasons that may lead to the failure of JIT compilation (e.g. syntax error in script or upper limit of JIT code heap size). Before searching for the target value, we will search for the magic value first. If the magic value is found in the potential code heap, we believe that the script code is compiled into JIT code. If the script code is not compiled into JIT code, there is no need to search for the target value. Another advantage of searching for the magic value is that the magic value is always located in the same code heap with JIT code of test script. We can filter out false positive errors if the target value is embedded in JIT code but the magic value is not. The generalized algorithm for searching non-blinded constant in Chrome is given in Algorithm 5.

4.4.2 Design of Minimizer

Minimizer has two jobs in our framework. First of all, minimizer has to check that the code is not false positive errors. If the monitor finds that the target value appears

Algorithm 5 Searching Algorithm for target value in Chrome

```
1: Input: targetValue
2: Output: whether the target value is JITted into code heap
3: heapInfo = getHeapInfo();
4: for all heap ∈ heapInfo do
5:     if getProtFlag(heap) == RWX then
6:         if valueExist(heap, 0xdead) or valueExist(heap, 0xbeef) then
7:             return valueExist(heap, targetValue)
8:         end if
9:     end if
10: end for
11: return false;
```

in the JIT code, the minimizer will load the target script in target application again and let monitor verify if the target value still exist in compiled JIT code. Secondly, the minimizer has to reduce the size of target script while keeping the target value appearing in JIT code. The minimizer will try to randomly remove a number of lines in the target script and load the script in target application again. If the target value still exist in JIT code, we suppose that the removed codes are redundant and repeat the previous step. If the target value or magic value no longer appears in the JIT code heap of rendering process, we restore the removed codes and repeat the steps above.

The minimizer will first scan the dumped file by monitor and locate the generated code according to the script template given in Section 4.3.3. To avoid causing syntax error in the process of removing lines of codes, the minimizer only removes the elements generated via one production rule in one round of minimization.

The Minimizer is not designed to give a minimum proof-of-concept (POC) of non-blinded constant but reduce the irrelevant codes as much as possible. We still need manual work to finally categorize the identified cases.

4.5 Implementation

In Fuzzer, the code generator is built on *jsFunFuzz* [106], which is heavily used for JavaScript fuzzing. Because *jsFunFuzz* is a grammar based fuzzer that aims at discovering memory-corruption bugs in JavaScript interpreter, the generation process in *jsFunFuzz*, i.e. makeStmt and makeExpr, will generate totally random code for fuzzing. Since our goal of the paper is to discover non-blinded constant in JIT code, we make a few customization in our code generator.

First of all, we have to reduce the syntax errors in the input script. We remove the codes that generate totally random script and modify the generation policy to make syntax errors less likely to occur. Next, we use node.js [66] to invoke the code generator and save the generated input script in a local file. In *jsFunFuzz*, the fuzzer will use *eval*(*code*) function to run generated code. However, size of the script generated in our tool always exceeds the upper limit of available memory for *eval*() function. To tackle this problem, we choose to insert the generated script into fuzzing template and save to a local file. The fuzzer then feeds the input file to target application.

The monitor is written in Python and consists of two parts. One part decides which process to attach. Another part searches for target value based on Algorithm 5. The second script is built on PYKD module [32], which is a python extension that provides users with access to the debug engine of WinDBG. After the target application loads input script, the first part gets the information of the running processes at present, including pid and image name, attaches debugger to the process and loads the second part to search for the magic value and target value.

The fuzzer and monitor communicate with each other via a TCP connection. When the fuzzer finishes generating the input file, the fuzzer sends a message to monitor and loads file into target application. After the monitor received the message from fuzzer, the monitor waits for browser to completely load the input file and generate the JIT code before attaching to running process. After searching through the address space of rendering process, the monitor terminates the running process

of target application and sends a message to fuzzer to start the next round.

4.6 Evaluation

To test the performance of our tool, we run our tool on Windows 10 in a VirtualBox virtual machine with Intel Core CPU i7-5557 having 3.1GHz and 8 GB RAM. We pick Chrome 58.0.3029.96 and Edge 38.14393 as our target application in our test. We run Blockade on Chrome for 12 hours and find 115 cases in total. On Edge, we run Blockade for one week and discover 21 cases in total. As explained in Section 4.3.3, for the same piece of code, Chrome will compile the code into JIT code at the first time running on it, Edge will run on the script multiple times before generating JIT code. In our test we choose to run the input code 200 times before searching for target value. Therefore we have to run our tool on Edge much longer than Chrome.

After minimizing the discovered cases, we further manually review the script code and categorize them according the the instruction, in which the non-blinded constant is embeded, as in Table 4.1. In this table, we find that array offset, object field, global variable and number of statement in script (in the following we will use statement number for simplicity) can all be used for generating non-blinded constant in JIT code. We further compare our result with previous work.

In the following part of this section, we are going to give the POC scripts to demonstrate the identified cases of our tool. All the POC scripts given in this chapter can be fetched via google site (`https://sites.google.com/view/blockadegrammarbasedapproach/home`) for interested readers to verify.

4.6.1 Array Offset

In Listing 4.3 and 4.4, we display the POC scripts that use the array index to release non-blinded constant in JIT code. In Chrome, the array offset is not protected by constant blinding, we find that the index number is directly JITted into the code heap.

	Browser	Call Offset	Constant	Array Offset	Object Field	Global Variable	Statement Number
Maisuradze et al [89]	Chrome	✓	✗	✗	✗	✗	✗
	Edge	✗	✗	✗	✗	✗	✗
Dachshund	Chrome	✗	✓	✗	✗	✗	✗
	Edge	✗	✓	✗	✗	✗	✗
Blockade	Chrome	✓	✓	✓	✓	✓	✓
	Edge	✗	✓	✓	✓	✓	✗

Table 4.1: Result Compared with Previous Work

```
arr = new Uint8Array();
fun_offsetr9()
{
  for(i=0; i<200; i++){
    arr[0x4141] = arr[0x4141] + 0x42;
    arr[12802368] = 0x43;
  }
}

JIT code:
mov eax, dword ptr [r14+0x20]
cmp eax, 0xc35941
jl Error
mov rax, qword ptr [r14+0x38]
mov ecx, 5758FA7C
xor ecx, 579BA33C
mov byte ptr [rax+rcx], 0x43
```

Listing 4.3: Chakra Array Offset POC

```
arr = new Uint8Array();

fun_offsetr9()
{
  for(i=0; i<20000; i++){
    arr[12802369] = 0x43;
  }
}

JIT code:
mov eax, 43h
mov byte ptr [rbx+0xc35941], al
```

Listing 4.4: Chrome Array Offset POC

For Edge, we think that the index number has been taken into consideration in its implementation of constant blinding as a patch for the case discussed in Listing 4.1. We find that the index number is XORed with a secret cookie. However, before writing data into the array there exists a checking instruction between `0xc35941` and member variable of an object. We think the instruction is to compare the index with the buffer capacity to prevent out-of-bound write. In our poc script, it is `12802368` (`0xc35940`) that is used as array index, but it is `0xc35941` that appears in JIT code.

In the work of Dachshund, it only finds cases where non-blinded constant number in JIT code is the same as the number inserted in script code. However, our discovery indicates that to make a specific number appear in JIT code, attacker does not need to use the same explicit constant in script code. For this reason, we separate the column of array offset from the column of constant number in Table 4.1.

Here we have to clarify that the loop count 20000 in Listing 4.4 is not the *LOOP_COUNT* given in the fuzzing template. The loop count in this case is generated via the naive algorithm in Section 4.3.2 . We set the loop count to be 20000 after some manual efforts of minimization. In Section 4.7.2, we will give a more detailed explanation about JIT compilation and how it influences our fuzzing result.

```
o = new Object();
o.a0 = 1;
...
o.a13000000 = 1;
funObject_r8r9()
{
  o.a12800762 = 0x4242;
  o.a12801018 = 0x4242;
}

JIT code:
mov     dword ptr [rsp+0x20], 0xc35841
mov     r9,rbx
xor     r8d,r8d

... /* similar JIT code*/
mov     dword ptr [rsp+0x20], 0xc35941
```

Listing 4.5: Edge Object Field POC

```
var a0=1;
var a1=1;
..
var a13000000 = 1;
funVariable_r8r9()
{
  a12800237 = 0x4242;
  a12801981 = 0x4242;
}

JIT code:
mov     dword ptr [rsp+0x20], 0xc35841
mov     r9,rax
xor     r8d,r8d

... /* similar JIT code*/
mov     dword ptr [rsp+0x20], 0xc35941
```

Listing 4.6: Edge Global Variable POC

4.6.2 Object Field and Global Variable

In Listing 4.5 and 4.6, we demonstrate how we utilize object field and global variable to release non-blinded constant in Edge browser.

In Listing 4.5, we define a global object o and assign a value to its member field. In function $funObject\_r8r9$, we reassign values to two of the object fields and run the function 200 times. With some efforts of reverse engineering on function discoved in JIT code, the non-blinded constant 0xc35941 is used as an index to fetch data in memory later. Similarly, POC code in Listing 4.6 demonstrates how global variable in script releases non-blinded constant in JIT code.

In both cases, the number in object field name and the number in variable name are not necessary conditions for releasing non-blinded constant in JIT code. We use numbers in the name to give a clear view about the structure of POC code. Attackers can use different names without numbers to get the same result.

As for Chrome browser, we also detect the index-like constant number in the JIT code. But such situation has been taken into consideration and the constant number is randomized by JIT compiler. In some of our fuzzing tests, we can allocate large amounts of object fields or variables and access all of them . With such a brute force method, the target value will be compiled into JIT code with a large probability. However, we do not take it as a stable way to generate non-blinded constant.

4.6.3 Statement Number

Another interesting discovery about the implicit non-blinded constant number is the statement number. In Listing 4.7 and 4.8, we list two POC codes to generate 0xc35941 and 0xc35841 in JIT code of Chrome. These two POC codes are manually minimized to find the minimum number of lines to release target value in JIT code.

Different from the relative call/jump offset that has been discussed before, the value of non-blinded constant will not be influenced by nop sledding. According to our analysis, the value is only linearly related with the statement number in the script.

```
funLine_r9()
{
        arr[1] = arr[1] + 1;   //1st line
        .
        .
        .
        arr[1] = arr[1] + 1; //2560473th line
}

JIT code:
mov rdi, 0xc3594100000000
```

Listing 4.7: Chrome number of statements POC that generates 0xc35941

If we append a new assignment statement $arr[1] = arr[1] + 1$ at the end of script in Listing 4.8, the instruction $mov\ rdi,\ 0xc3584100000000$ still exists in JIT code and a new instruction $mov\ rdi,\ 0xc3584600000000$ will be emitted in code heap following the previous one.

4.6.4 Relative Call Offset and Constant Number

In our search, cases where relative call offset and explicit constant number are utilized to emit target value are also discovered by our tool. In the following, we will give a brief demonstration about them respectively.

Relative Offset In JIT code of the POC in Listing 4.8, we discover more than one sites where the target value is embedded in the relative offset of the indirect call instruction. We list one instruction discovered in Chrome, where target value (0xc35841) is embeded in the relative offset.

Explicit Constant For explicit constant number that evades constant blinding, Blockade can also find cases both in Chrome and Edge.

In Listing 4.9, an explicit constant number in the comparison expression of a for loop is not protected by constant blinding. In Listing 4.10, an explicit constant number in the argument of an user-defined function. In Dachshund, the author emphasizes that arguments to the built-in function of MATH will not be blinded. But we

```
funLine_r8()
{
        arr[1] = arr[1] + arr[1];
        arr[1] = arr[1] + arr[1];
        arr[1] = arr[1] + arr[1];
        arr[1] = arr[1] + arr[1];
        arr[1] = arr[1] + arr[1];
        arr[1] = arr[1] + arr[1];
        arr[1] = arr[1] + arr[1];

        arr[1] = arr[1] + 1;  //1st line
        .
        .
        .
        arr[1] = arr[1] + 1; //2560412th line
}

JIT code:
mov rdi, 0xc3584100000000
```

Listing 4.8: Chrome number of statements POC that generates 0xc35841

```
funConstant_r9()
{
  for(var i=0; i<0xc35941; i++) a = a+1;
}

JIT code:
xor eax, eax
cmp eax, 0xc35941
jge branch_code
```

Listing 4.9: Chrome Explicit Number POC

```
a(t) {return t+1;}
funConstant_r9()
{
  for(var i=0; i<200; i++)  a(0xc35941);
}

JIT code:
mov rdi, 0x1000000c35941
```

Listing 4.10: Edge Explicit Number POC

Browser	Relative Offset	Explicit Constant	Array Offset	Object Field	Global Variable	Line Number
Chrome	5.325s	0.126s	0.188s	-	-	13.188s
Edge	-	0s	0.593s	50.586s	67.751	-

Table 4.2: Loading time of Fuzzer for the POC script

Browser	Relative Offset	Explicit Constant	Array Offset	Object Field	Global Variable	Line Number
Chrome	1.327s	0.354s	0.028s	-	-	4.422s
Edge	-	0.251s	0.292s	10.140s	6.203s	-

Table 4.3: Searching time of Monitor for the POC script

find that arguments of user-defined function will not be blinded by constant blinding either.

In Edge, a piece of code will not be compiled into JIT code until the code has been executed several rounds. To further clarify the cases of non-blinded explicit constant in Edge, we use the built-in function of MATH module to test the threshold for JIT compilation. In our test, we find that the script code will not be compiled into JIT code until the function has been executed over 150 times. However, the threshold set in Dachshund is 50. Such difference shows the execution threshold of JIT compilation on Edge may vary on different systems. Therefore it's reasonable to set $LOOP\_COUNT$ to 200 in our fuzzing template to assure the code will be compiled into JIT code in Edge.

Though the two POC scripts given in Listing 4.9 and 4.10 are categorized as explicit constant, they are not disclosed by Dachshund. Because explicit constant number is not covered by our grammar-based approach, we list the cases in this section to prove our fuzzing framework is well-designed and complete enough to discover the cases found in previous work.

4.6.5 Performance Evaluation

To evaluate if our attack is practical in real world, we evaluate how long it takes from loading script to emitting target value in JIT code. To do so, we insert the built-in function *getTime*() of DATE into the POC scripts given above to get running time. We run each script code 10 times and get the average time. In Table 4.2, we display the running time of the POC scripts given above on Chrome and Edge. All the time lengths given in the table are the time needed to emit 0xc35941 in JIT code. In the table 0s denotes the time length is short and negligible.

We further test the time needed of our monitor to search for the target value in script rendering process. To log the time in debugging session, we use the !*time* in debugger to get searching time. In Table 4.3, we give the searching time in JIT code of the POC scripts given above.

Above all, the average of running time on Edge is longer than Chrome. The result is reasonable because we run the test code 200 times in Edge. However after minimization of the identified case, the running time on Chrome and Edge are acceptable for a real-world exploit.

In case of relative jump offset and statement number, minimizer can remove 30.2%-56.5% of lines in the file dumped from monitor. However, minimizer is hard to remove lines in case of object field and global variable. Any deletion of the object field or global variable will influence the JIT code generated in the final. In future, a smarter minimizer is required not to remove the declaration part of object or variable. At present, we rely on manual work to identify the specific object field or global variable that emits target value in JIT code.

4.7 Discussion

In this section, we will discuss the implication of our work. For some non-blinded constants in JIT code identified by Blockade, we give a more detailed explanation

from the view of JIT compilation. We further propose potential defence based on the cases discovered in this chapter and discuss limitation of them.

4.7.1 Implication

Web browser has always been a popular target for exploitation. Script language, e.g JavaScipt and ActionScript, supported by modern browser provide attackers multiple attacking vectors in exploit generation. Recent researches [49, 112] give POC exploits based on JIT spray and prove that the threat from JIT spray is real and practical.

Before our work, there is no systematic approach to discover non-blind constant in JIT code. Athanasakis et al. [44] only focus on 2-byte value in JIT code, which provides little flexibility for choosing gadgets in exploitation. Maisuradzu et al. [89] propose the concept of implicit constant in their paper, but they only find that relative jump/call offset can be used as non-blinded implicit constant. The relative offset in indirect jump/call instruction are also influenced by nop sledding. Dachshund [90] proposed a fuzzing framework to search for explicit constant number that is not covered by constant blinding. Based on previous work, we propose a grammar-based fuzzing framework and find more cases that have not been discovered before. Our work shows that the structural information based on different grammar can also be used to discover non-blinded value in JIT code. All the cases of implicit constant identified by Blockade will not be influenced by nop sledding. Since our approach is based on grammar of a script language, we can apply the same approach on ActionScript (Flash) to evaluate the constant blinding mechanism in it. To sum up, our work does not concentrate only on one type of occasion where constant blinding fails to cover. We provide a more general approach for searching non-blinded constant in the JIT code.

Script Code	JIT Code
a = a + 1; arr[0xc35940] = 0x43	cmp [rax+20h],0xc35940 jle CodeBranch

arr[0x4141] = arr[0x4141] + 1;
arr[0xc35940] = 0x43
	mov eax,[r13+20h]
	cmp eax, 0xc35941
	jl CodeBranch

Table 4.4: JIT code on Chakra

4.7.2 JIT Compilation

In this section, we are going to discuss two important observations about JIT compilation after having analysed the fuzzing result.

In the POC code of Listing 4.3, we give a example code where explicit constant in JIT code does not have to be identical to the number inserted in script. To dig more information from this sample, we do some tests on the JIT code with the different combination of statements. In Table 4.4, we list the test script code and corresponding generated JIT code. For script code `arr[0xc35940]=0x43`, JIT compiler generates two different but semantically equivalent JIT codes.

Moreover, such JIT code does not only exist in the case where the type of *arr* is Uint8Array. We have verified that such JIT code still exists in case of Uint16Array, Uint32Array and Float64Array. Therefore, we suppose that the one-byte difference in JIT code is irrelevant with the data type in the array but results from the code optimization in JIT compilation.

From the POC code in Listing 4.4, we found that there exist two stages of JIT compilation in Chrome. In the first stage, the compiler will compile the script into native JIT code. If the times that script has been executed exceeds a threshold , the JIT compiler will further optimize the generated JIT code to improve the performance.

In Table 4.5, we list the different JIT codes of `arr[0xc35941]=0x43`, which are generated in different stages of JIT compilation in Chrome. To trigger the first stage

First Stage	Second Stage
mov r10,0x231c56fa push r10 mov r10 0x23df0fbb xor [rsp], r10	mov [rbx+0xc35941],al

Table 4.5: Stages of JIT Compilation on Chrome

compilation on Chrome, the loop count is 1. To trigger the second stage compilation on Chrome, the loop count is 20000. More importantly, for the same piece of code, constant blinding is implemented in the first stage compilation but missed in the second stage. According to our analysis, the threshold of loop count for the second stage is not a fixed number.

For the the same piece code, both irrelevant statements and JIT compiler optimization may influence the fuzzing result. In future, a more adaptive generation approach is required to take all such information for fuzzing test.

4.7.3 CFI enforcement in JIT code

The best approach to defend against JIT Spray attack is to apply CFI enforcement in JIT code. Even if attackers can somehow generate the desired byte sequence in JIT code, CFI enforcement can still block the illegal control flow transfer to those gadgets. JITScope [132], RockJIT [97], NaCl-JIT [43] propose various JIT enforcement policy in JIT code.

In JITScope, each dynamically generated function is assigned an unique ID and instruments the ID before the function. Furthermore, JITScope deploys W$\oplus$X enforcement in JIT code to prevent attackers crafting fake ID in code region. The frequent calls to API $VirtualProtext$ bring extra performance in the benchmark evaluation. How to achieve a balance between security enforcement and performance is still a challenging work in future.

Chapter 5

Layered Object Oriented Programming

5.1 Introduction

Very recently, vtable reuse attack, exemplified by counterfeit object-oriented programming (COOP) [108] emerges as a novel type of code reuse attacks. COOP attacks can bypass most of the binary-level CFI enforcement as well as the vtable integrity enforcement (e.g., T-VIP [76] and VTint [133]). So far, multiple source-level solutions [65, 91, 131] have been developed to defend against vtable reuse attacks. Given that source code is not always available, a binary-level solution, TypeArmor [124], is recently proposed to defend against COOP attacks. Besides, vfGuard [103] can limit the valid targets in vtable reuse attacks. Specially, TypeArmor and vfGuard respectively use argument register count and dispatch offset at virtual callsite as the signature for checking the validity of target functions.

In this chapter, we revisit the most advanced binary-level mitigation (i.e., TypeArmor [124] and vfGuard [103]) against vtable reuse attacks, and propose *Layered Object-Oriented Programming* (LOOP), a more sophisticated and advanced vtable reuse attack than COOP, to bypass both TypeArmor and vfGuard. In par-

ticular, we propose *argument expansion gadget* that can expand argument count and load desired value into argument registers through explicit data flow without violating the CFI policy of TypeArmor, and *transfer gadget* that can modify the dispatch offset used at virtual callsite to match the offset of target virtual function for evading vfGuard. Moreover, we design and implement tools to discover both gadgets in the binary code of victim programs using forward/backward data flow analysis.

To demonstrate the feasibility of LOOP and the wide availability of potential gadgets in real-world complicated applications, we launched LOOP attacks on three real-world applications: Firefox, Flash Player and Internet Explorer. We discovered sufficient number of gadgets of both types in these three applications, and presented the proof-of-concept (POC) attacks based on existing CVEs.

As an extended variation of vtable reuse attacks, this work defines and discovers more types of gadgets for vtable reuse attacks. We prove that those coarse-grained CFI systems, simply matching argument count and offset into vtable, are insufficient to defend against sophisticated vtable reuse attacks with delicately chosen gadgets and carefully prepared memory layout. In this chapter, we make the following contributions.

- We propose Layered Object-Oriented Programming (LOOP) and introduce the concept of argument expansion gadget and transfer gadget, which can bypass the most advanced binary-level mitigations (i.e., TypeArmor and vfGuard).
- We design and implement tools to discover argument expansion gadgets and transfer gadgets at the binary level to construct a successful exploit.
- We present a POC exploit for Firefox on the Linux platform and for Flash Player and Internet Explorer on the Windows platform. All of these exploits are based on existing vulnerabilities, and we construct complete exploitation chains.

5.2 TypeArmor and VFGuard

5.2.1 TypeArmor

TypeArmor [124] incorporates a CFI strategy to enforce that each callsite strictly targets matching functions only. In particular, the CFI ensures that indirect callsites that set at most *max* arguments cannot target functions that use more than *max* arguments (but can target functions that use no more than *max* arguments). Besides, the CFI ensures that indirect callsites that expect a return value (i.e., non-void callsites) cannot jump to a callee that does not prepare such value (i.e., void functions).

Additionaly, TypeArmor incorporates a novel Control Flow Containment (CFC) strategy to disrupt type-unsafe function argument reuse attempts. Specifically, TypeArmor scrambles the unused arguments at every indirect callsite before transferring control to the callee such that illegal function targets are not inadvertently exposed to stale arguments. Similarly, TypeArmor scrambles unused return arguments before transferring control back to the caller.

To be more specific, TypeArmor relies on forward and backward analysis to obtain argument register usage information of callsite and callee functions. We will take the assembly code of a function in Listing 5.1 for a more detailed explanation.

Forward Analysis (Callee Analysis). In forward analysis of TypeArmor, the status of a register can be: (1) read-before-write (R), (2) write-before-read (W), and (3) clear (C), where R represents that the register is always read before a new value is written into; W represents that a value is always written into the register before the register is read; and C represents that the register is never read or written into.

To determine the number of used argument registers at the callee side, TypeArmor conducts a recursive forward analysis from the beginning of the function. For direct calls and returns, TypeArmor maintains a stack to emulate the execution. For indirect calls, TypeArmor conservatively assumes that the target writes all argument registers, and stops the recursion. Once the recursive analysis converges, the argument count

```
1  soundtouch::FIFOSamplePipe::moveSamples(soundtouch::
       FIFOSamplePipe&)
2  push    r13
3  push    r12
4  mov     r12, rdi
5  mov     rdi, rsi
6  push    rbp
7  push    rbx
8  mov     rbx, rsi
9  push    rax
10 mov     rax, [rsi]
11 call    qword ptr [rax+0x30]
12 mov     ebp, eax
13 mov     rax, [r12]
14 mov     rdi, rbx
15 mov     r13, [rax+18h]
16 mov     rax, [rbx]
17 call    qword ptr [rax+0x10]
18 mov     edx, ebp
19 mov     rsi, rax
20 mov     rdi, r12
21 call    r13
22 mov     rax, [rbx]
23 mov     rdi, rbx
24 mov     esi, ebp
25 mov     rax, [rax+0x28]
26 pop     rdx
27 pop     rbx
28 pop     rbp
29 pop     r12
30 pop     r13
31 jmp     rax
```

Listing 5.1: Sample Binary Code of Firefox on Linux for Forward Analysis and Backward Analysis in TypeArmor

is set using the highest argument register that is marked as R. Note that, on 64-bit platform, Application Binary Interface (ABI) defines an argument passing sequence: on Linux, the passing sequence is `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` [34]; on Windows, the passing sequence is `rcx`, `rdx`, `r8`, `r9` [37].

For example, before the first indirect call at Line 11 in Listing 5.1, there is a read operation on `rdi` and `rsi` at Line 4 and 5 respectively. Therefore, the argument

register number of the function `moveSamples` will be set to at least 2.

Backward Analysis (Callsite Analysis). In backward analysis, the status of a register can be: (1) set (S) and (2) trashed (T). To determine the number of prepared argument registers at the indirect callsite, TypeArmor performs a backward analysis, starting from the basic block that contains the callsite. If there exists an incoming edge to the current basic block, TypeArmor continues the backward analysis along the incoming edge. As there is no incoming edge for indirect call targets, the analysis terminates if the callsite is in a virtual function or a function call that is only used as a function pointer. Further, TypeArmor scrambles unused registers to prevent an attacker from loading data of previous function calls into argument registers. Similar to the forward analysis, once the recursive analysis is finished, the number of prepared arguments is set based on the status of the last write operations.

For example, before the indirect call at Line 21 in Listing 5.1, values are written into `rdi`, `rsi`, and `edx` (the lower 32 bit of `rdx`). Therefore, the callsite at Line 21 prepares at most 3 argument registers. For those unused registers `rcx`, `r8` and `r9`, TypeArmor assigns a random number to them.

5.2.2 vfGuard

For virtual function dispatch, there are two implicit assumptions: (1) the assembler is assumed to generate the correct dispatch offset in the assembly code for fetching function pointer in vtable; (2) the corresponding function pointer is assumed to be placed at the correct offset in vtable. In a benign execution of a virtual function call, these two offsets are actually two identical constant values.

Based on these two implicit assumptions, vfGuard [103] constructs its enforcement strategy. In particular, the valid targets of a virtual function call is restricted to the set of virtual functions whose vtable offset is equal to the dispatch offset at the callsite. Hence, vfGuard ensures that the vtable pointer of an object always points to the top of a vtable.

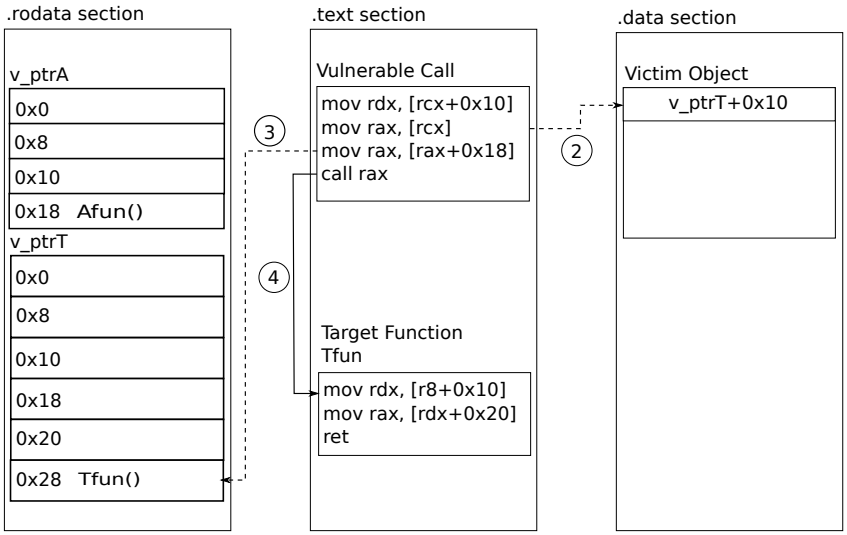


Figure 5.1: Blocking COOP by TypeArmor and vfGuard on Windows (The Dashed Line Denotes Data Fetch and the Solid Line Denotes Control Flow Transfer.)

To get the dispatch offset at the binary level, vfGuard conducts a backward analysis, starting from the callsite. After locating the dereference operation that fetches the function entry from vtable, vfGuard extracts the dispatch offset from assembly code. To discover the vtable offset of the function pointer, vfGuard applies a few heuristics to locate the top of the vtable in memory. Once the top of the vtable is determined, the vtable offset of the function pointer is determined accordingly. Notice that we take similar heuristics to scan for vtables in memory, and more details will be discussed in Section 5.4.1.

5.2.3 TypeArmor and vfGuard Against VTable Reuse Attack

Both TypeArmor and vfGuard can limit the set of valid targets, using different semantic information at the binary level. Fig. 5.1 illustrates how TypeArmor and vfGuard are expected to mitigate COOP. The dispatch offset of virtual callsite is `0x18` but the vtable offset of the target function pointer is `0x28`, attackers have to overwrite the vtable pointer of the victim object with `v_ptrT+0x10`. In the virtual function dis-

patch (Step 3), such function pointer fetch will be blocked by vfGuard as the vtable pointer of the victim object does not point to the top of the vtable. In Step 4, the argument count at the callsite is identified as 2 (i.e., `rdx` is marked as S), while the argument count for the target function is identified as 3 (i.e., `r8` is marked as R), such control flow transfer will be blocked by TypeArmor as the target function requires more arguments than the callsite. While being the most advanced defenses against vtable reuse attacks, TypeArmor and vfGuard are still vulnerable to vtable reuse attacks, as will be proved in Section 5.3 and 5.4.

5.3 Overview of Layered Object Oriented Programming

In this section, we first introduce our threat model and our attacking goal for LOOP before we give an overview of LOOP. Then we motivate the choice of two types of gadgets, argument expansion gadget and transfer gadget, with code examples, and explain how these two types of gadgets are expected to bypass TypeArmor and vfGuard. Finally, we show how these gadgets are chained together for a successful exploit.

5.3.1 Threat Model and Attacking Goal

Threat Model. To demonstrate our exploit against TypeArmor and vfGuard, we assume that the target binary follows the conditions below. Notice that these assumptions conform to the attacker settings of most defenses against vtable reuse attacks.

- The target is under the protection of DEP [5] and ASLR [31] which are often enabled at the hardware and OS level.

- TypeArmor and vfGuard are simultaneously deployed in the target to defend against code reuse attacks.
- The target is vulnerable to information leakage [111]. Hence, the attacker can read arbitrary value in memory and identify all the vtables and the corresponding virtual functions.
- The target is vulnerable to memory corruption. Thus, the attacker can craft or modify the member variable of an object in writeable area.

Attacking Goal. LOOP attacks start from a virtual function callsite with no argument prepared, which is actually the most tough case to exploit under the protection of TypeArmor. The goal of our attacks is to call *mprotect* (on Linux) or *VirtualProtect* (on Windows) to make an area of memory under attackers' control executable for further exploits. From our experience on writing exploits, if attackers are able to successfully change the protection flag of desired memory, attackers are able to execute shellcode afterwards. Moreover, for critical system calls (e.g., *WinExec* and *execv*) that require less arguments than *mprotect* and *VirtualProtect*, if we can call *mprotect* and *VirtualProtect* with arguments under our control, we can call those functions in the same way.

5.3.2 LOOP Overview

LOOP is built upon the observation that some objects contain a pointer that refers to another object, which makes it possible to arrange crafted objects in the memory to divert control flows for invoking critical system calls. As shown in Fig. 5.2,for the *victim object* of class `A` in *.data* section, the attacker corrupts the member variable `o1` (`corrupted ptr_o1`), and attempts to invoke the virtual function `Afun`. The corrupted member variable is then used as counterfeit object and argument for the virtual callsite in `Afun`, and the control flow is diverted to `BTRfun`. Similarly, by carefully crafting the member variable `o2`, `o3`, `o4`, `o5` and `buf` of the object of class

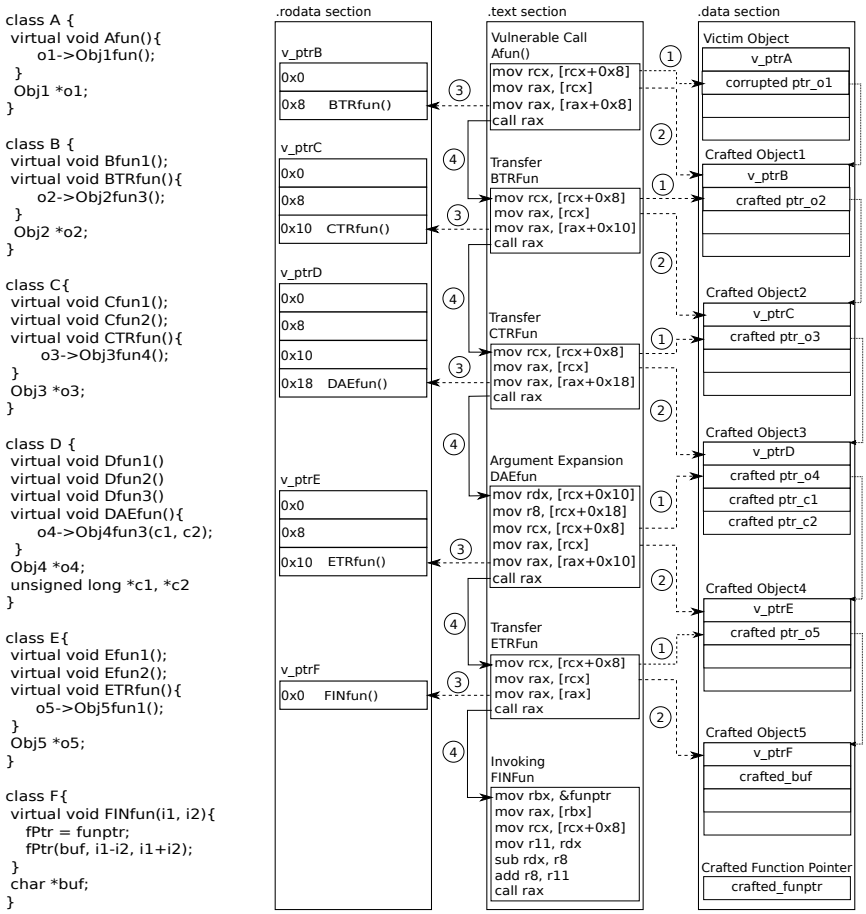


Figure 5.2: An Example of Layered Object-Oriented Programming on Windows (The Dashed Line Denotes Data Fetch, the Solid Line Denotes Control Flow Transfer, and the Dotted Line Denotes Point-To Relation.)

`B`, `C`, `D`, `E` and `F`, the attacker finally diverts the control flow to `FINfun`, where the attacker crafts the function pointer to invoke a critical system call. Since TypeArmor and vfGuard only check the validity of the target function at callsite but set no data integrity checking before callsite, corruptions on member variables will not be detected by both mitigations.

However, such a chaining may still be blocked by TypeArmor and vfGuard (as discussed in Section 5.2.1 and 5.2.2). Thus, we propose *argument expansion gadget* (Section 5.3.3) to expand argument count and load desired value into argument reg-

isters via explicit data flow without violating the CFI policy of TypeArmor, and *transfer gadget* (Section 5.3.4) to modify the dispatch offset used at virtual callsite to match the offset of target virtual function for evading vfGuard. Using these carefully chosen gadgets, we construct LOOP attacks (Section 5.3.5).

5.3.3 Argument Expansion Gadget

The control flow in Fig. 5.2 can be diverted from the *vulnerable call* `Afun` (in *.text* section) to `FINfun` by crafting the member variable `o1` and the vtable pointer `v_ptrA` of the *victim object* of class `A`. As the number of used argument registers (i.e., 3, including the register for the implicit `this` pointer) of `FINfun` is larger than the number of prepared argument registers (i.e., 1) at the virtual callsite of `Obj1fun`, the control flow is blocked by TypeArmor. However, TypeArmor misses the possibility that some wrapper functions can explicitly expand the argument count via execution flow, and we call such functions *argument expansion gadgets* (see Definition 5.3.1).

Definition 5.3.1. An argument expansion gadget is a virtual function which contains a virtual callsite that prepares more argument registers than the virtual function itself. The argument expansion gadget expands the argument register count from n_1 to n_2, where n_1 is the number of argument registers used by the function itself and n_2 is the number of argument registers prepared at the virtual callsite within the function.

In Fig. 5.2, the virtual function `DAEfun` has no arguments but contains a virtual callsite of `Obj4fun3` that prepares three argument registers. Further, `o4`, `c1` and `c2` are all member variables under attacker's control. Through crafting these member variables, the attacker can invoke the virtual function `DAEfun` to divert the control flow to virtual functions that requires two arguments, and keep the values in the three argument registers under control. Thus, `DAEfun` is an argument expansion gadget that expands the argument count from 0 to 2 (or expand the argument register count from 1 to 3).

Bypassing TypeArmor. Argument expansion gadgets in our LOOP attacks expand argument count via existing instructions, pass value to argument registers through explicit data flow, and divert control flow to target functions without being blocked by TyperArmor via crafting member variables and vtable pointers. For example, in Fig. 5.2, `Afun`, `DAEfun` and `FINfun` can be chained together by crafting member variables `o1` and `o4` and overwriting `v_ptrD` and `v_ptrF` with `v_ptrD+0x10` and `v_ptrF-0x10` respectively to match the vtable offset, which successfully bypasses TypeArmor.

Moreover, a gadget that expands argument count from n_1 to n_2 and a gadget that expands argument count from n_3 ($n_1 \leq n_3 \leq n_2$) to n_4 can be chained together to expand argument count from n_1 to n_4, because TypeArmor only blocks control flow transfers to functions that require more arguments.

5.3.4 Transfer Gadget

As illustrated in Section 5.3.3, chaining `Afun`, `DAEfun` and `FINfun` needs to respectively craft `v_ptrD` and `v_ptrF` to `v_ptrD+0x10` and `v_ptrF-0x10` to match the vtable offset. As these crafted vtable pointers do not point to the top of vtables, the function pointer fetch will be blocked by vfGuard. Possibly, the `this` pointer used to invoke a virtual function `f1` (whose vtable offset is `o1`) could be different from the `this` pointer used to call a virtual function `f2` (whose vtable offset is `o2`) in `f1`, and `o1` and `o2` could be different. This makes it possible to adjust the dispatch offset at virtual callsite through chaining virtual functions like `f1`, and we call such functions *transfer gadgets* (see Definition 5.3.2).

Definition 5.3.2. A transfer gadget is a virtual function whose vtable offset is different from the dispatch offset used at the virtual callsite within the function. The transfer gadget adjusts the dispatch offset from o_1 to o_2, where o_1 is the vtable offset of the function itself and o_2 is the dispatch offset used at the virtual callsite within the function.

In Fig. 5.2, the vtable offset of the virtual function `BTRfun` is `0x8`, while the dispatch offset at the callsite of `Obj2fun3` is `0x10`. Thus, `BTRfun` is a transfer gadget that adjusts the dispatch offset from `0x8` to `0x10`. Similarly, both `CTRfun` and `ETRfun` are transfer gadgets that respectively adjust the dispatch offset from `0x10` to `0x18` and from `0x10` to `0x0`.

Bypassing vfGuard. In order to match the vtable offset with the dispatch offset and thus not violate vfGuard, we insert multiple transfer gadgets between the vulnerable call and argument expansion gadget, between two argument expansion gadgets or between argument expansion gadget and the target function to chain them together. In Fig. 5.2, we show how transfer gadgets are used to divert control flow from the vulnerable call to the argument expansion gadget and from the argument expansion gadget to the target function. At the vulnerable callsite, the dispatch offset is `0x8`, while for the argument expansion gadget `DAEfun`, its vtable offset is `0x18`. In LOOP, we first divert the control flow from the vulnerable callsite to the transfer gadget `BTRfun`, whose vtable offset is `0x8` and matches the dispatch offset at the vulnerable callsite. Then we divert the control flow from the virtual callsite of `Obj2fun3` to the transfer gadget `CTRfun`, whose vtable offset matches the dispatch offset at the callsite of `Obj2fun3`. Finally, the control flow is diverted to the argument expansion gadget `DAEfun`, whose vtable offset matches the dispatch offset at the callsite of `Obj3fun4`. Similarly, the transfer gadget `ETRfun` is used to link the argument expansion gadget `DAEfun` and the target function `FINfun`.

Transfer Gadget Against TypeArmor. For a virtual function gadget that directly passes its arguments to a virtual call within the function, compiler optimization will omit those unnecessary instructions that set argument registers. As a result, the argument count will be falsely computed at the virtual callsite within such wrapper gadgets. To avoid potential false positives, TypeArmor assumes that such wrapper gadgets prepare the maximum number of argument registers at virtual callsite.

As an example, the function in Listing 5.2 is a transfer gadget, adjusting the dis-

```
1  CUnifiedListView::ActivateHighlightedItem(enum OLECMDID)
2  mov     rcx, [rcx+0x8]
3  mov     rax, [rcx]
4  jmp     qword ptr [rax+0x110]
```

Listing 5.2: A Sampe Code for Transfer Gadget in ieframe.dll

patch offset from `0x88` to `0x110`. It has only three binary-level instructions. According to the forward analysis and backward analysis of TypeArmor, the register `rdx` is not identified as R. Thus, this function should only take one argument register (i.e., the `this` pointer), and the virtual callsite could only target virtual functions with one argument register. However, from the function name derived from the debugging symbol, it actually takes two argument registers. The usage of the second argument register was optimized out by compiler.

To limit the potential impact of this conservative assumption, TypeArmor scrambles the values in unused argument registers at the callsite that invokes such wrapper gadgets in the hope that the scrambled values cannot be used for exploit generation; i.e., TypeArmor will not scramble argument registers whose value is set through explicit data flow. In LOOP, we use argument expansion gadgets to load value into argument registers, and pick such wrapper gadgets as transfer gadgets. Thus, the control flow transfer between argument expansion gadgets and transfer gadgets will not be blocked by TypeArmor.

5.3.5 Exploit Generation

Invoking Gadget. After preparing the values in argument registers, we seek invoking gadget (i.e., the target function) to call a critical system call. An invoking gadget is a virtual function gadget that invokes a function pointer, which comes from an argument register or memory under attacker's control.

Listing 5.3 shows an invoking gadget in ICU library, which we use in our exploit on Firefox (Section 5.5.1). Since function pointer invocation is not in the protection

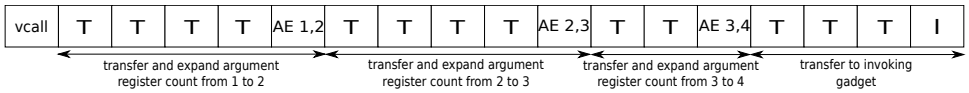


Figure 5.3: Gadget Chaining in LOOP (**vcall** Denotes a Vulnerable Virtual Call. **T** Denotes a Transfer Gadget. **AE** Denotes an Argument Expansion Gadget, Followed by Two Numbers Denoting Argument Register Counts. **I** Denotes an Invoking Gadget.)

```
void ReplaceableGlue::copy(int start,int limit,int dest) {
  (*func->copy)(rep, start, limit, dest);
}
```

Listing 5.3: An Invoking Gadget to Invoke Function Pointer in ICU Library

scope of vfGuard, there should be no vfGuard check on the validity of the target. As for TypeArmor, the arguments are either loaded from member variables or passed through callers. Therefore, TypeArmor cannot block the control transfer either.

Gadget Chain Construction. Different from a COOP attack, a LOOP attack chains gadgets through virtual function calls. Fig 5.3 gives an overview of how LOOP is expected to call a critical function. In the first step, we load value into argument registers through chaining argument expansion gadgets. In the second step, we invoke a critical system call through invoking gadget. To chain all these gadgets together, we find multiple transfer gadgets to adjust the dispatch offset at virtual callsite. In the view of memory layout, the crafted objects are chained via reference, as llustrated in Fig. 5.2. A latter object in the chain is always referenced by a pointer in the former object.

5.4 Gadget Discovery

In this section, we first introduce how to discover argument expansion gadgets, transfer gadgets and invoking gadgets from binary code and then present the tool implementation.

5.4.1 Identification of Virtual Functions

As argument expansion gadgets and transfer gadgets are virtual functions that contain at least one virtual callsite, we first introduce the identification of virtual functions, i.e., identifying the vtables where the virtual function pointers are located and computing the vtable offsets of virtual function pointers.

In particular, we design three heuristics to identify the candidate vtables in binary code. **H1**: the function address in the vtable must be the entry point of a function; **H2**: the vtable lies in a read-only region; and **H3**: only the beginning of the vtable is referenced in a text region.

H1 is used to filter out the jump tables of code pointers that jump into the middle of a function. **H2** is based on the observation that a vtable always locates in a read-only region. Specifically, we assume that a vtable locates in *.rodata.data.rel.ro* and *data.rel.ro.local* on Linux platforms, while a vtable locates in *.rdata* and *.text* on Windows platforms. We include the *.text* region as we find that some vtables locate in the *.text* region in *mshtml.dll* and *ieframe.dll*. **H3** is used to determine the top of a vtable. In constructor function of an object that contains virtual functions, a reference to the vtables is stored in the object. The top of a vtable is usually referenced in a constructor function. Therefore, we rely on this heuristic to determine whether the address is the top of a vtable or not.

Algorithm 6 shows the procedure to identify candidate virtual functions in a target program and compute their vtable offsets. To satisfy **H1**, it obtains and iterates over all the functions in the target binary (Line 4–5). As a virtual function may exist in multiple vtables, it gets all the addresses where a function is located (Line 6), and iterates over all potential vtables to find all possible offsets (Line 7–21). If the address is not in a read-only region, it applies **H2** to filter it out (Line 8–10); otherwise, to locate the top of a vtable, it tries to find the least vtable offset that makes the address satisfy **H3** (Line 11–20). Specifically, it begins the searching from offset 0, and increases the offset by 8 in each iteration. If the offset exceeds a threshold (e.g., 0x400 in

our evaluation), we assume that it is a global function pointer and stop the searching.

Notice that our determination of the top of a vtable is different from vfGuard. vfGuard scans read-only region and accumulates the valid function pointer until it encounters an invalid address. Thus, vfGuard will not stop its backward search until it finds an invalid function pointer. On the contrast, we utilize the constructor function in *.text* region to locate the top of a vtable since the constructor function always contains a reference to the top of a vtable. The root cause of such differences lies in the different purposes of LOOP and vfGuard. vfGuard is to provide protection against vtable reuse attacks, and can tolerate some false positives in vtable identification so as to cover as many potential vtables in the binary as possible. However, LOOP is to bypass binary-based mitigation, and missing some vtables is tolerable as long as LOOP can discover enough true vtables for launching vtable reuse attacks.

5.4.2 Discovery of Argument Expansion Gadgets

We apply three conditions to search for argument expansion gadgets. **AC1**: the function is a virtual function; **AC2**: the function contains a virtual callsite that prepares more argument registers than the function itself; and **AC3**: the argument register values at the virtual callsite in the function are under attacker's control. In particular, for each candidate virtual function identified by Algorithm 6 (i.e., satisfying **AC1**), we apply the same forward and backward analysis as in TypeArmor to detect gadgets satisfying **AC2**, and perform forward data flow analysis to determine whether the candidate virtual function satisfies **AC3**.

Algorithm 7 shows the general procedure to decide whether a candidate virtual function returned by Algorithm 6 is an argument expansion gadget. It first uses the forward analysis of TypeArmor to get the arguments of the candidate virtual function (Line 3). Then, it constructs the control flow graph (CFG) of the candidate function and collects all paths in CFG (Line 4). For each collected path, we perform our forward data flow analysis from the beginning of the function (Line 5–17).

Algorithm 6 Identify Candidate Virtual Functions

```
1: Input: P: the binary code of a target program
2: Output: VF: candidate virtual functions and their vtable offsets
3: VF = ∅
4: F = the set of functions in the target program P
5: for each function f in F do
6:     ADDR = getRef(f)
7:     for each address addr in ADDR do
8:         if not inReadOnlyRegion(addr) then
9:             continue
10:        end if
11:        offset = 0
12:        while offset < threshold do
13:            refToAddr = getRef(addr − offset)
14:            for each ref in refToAddr do
15:                if inTextRegion(ref) then
16:                    VF = VF ∪ {< f, offset >}
17:                end if
18:            end for
19:            offset = offset + 8
20:        end while
21:    end for
22: end for
23: return VF
```

In our forward analysis, it uses a set *ctrl_set* to hold argument registers whose values are under attacker's control. Based on our threat model, the attacker is assumed to be able to modify member variables of vulnerable objects. Therefore, this set initially contains the `this` pointer which is passed through the first argument register (Line 7). On Linux platform, *ctrl_set* is initialized to contain the `rdi` register (Line 7). On Windows platform, *ctrl_set* is initialized to contain the `rcx` register.

For each virtual callsite in a path (Line 8–9), it first uses the backward analysis of TypeArmor to get the argument registers at the callsite (Line 10). If all these

argument registers belong to *ctrl_set* and the number of these argument registers is larger than the number of arguments of the candidate virtual function (satisfying **AC3** and **AC2**), the candidate virtual function is regarded as an argument expansion gadget (Line 11–13).

In our forward analysis on each path, it updates *ctrl_set* according to the instructions in each path (Line 15). Specifically, we generalize three cases that can load a desired value to an argument register via explicit data flow.

- **Member Variable.** The virtual callsite in the candidate virtual function uses the member variable of *this* object as the function argument. We add the register that holds the `this` pointer to *ctrl_set*. If a member variable of a victim object is loaded into a register via `[this+offset]`, we also add the register to *ctrl_set*.

- **Return Value.** The virtual callsite uses the return value of a previous virtual call as the function argument. To track the return value of the previous virtual call, we add the `rax` register that holds return values to *ctrl_set* after a virtual function call, and continue our forward analysis. At the next virtual callsite, we check whether its argument registers are in *ctrl_set* or not. Here, we assume that every virtual callsite is a non-void callsite, which calls a non-void function and expects a return value. In our analysis, we check between every two consecutive virtual calls and judge whether the last one uses the return value from the previous one as its argument. If this is true, there must exist a read-before-write operation on `rax` to load the value into the argument register, which proves that `rax` saves the return value of the previous virtual call. If not, the value in `rax` will not be passed to argument register to influence our result. Hence, such an assumption is reasonable in our analysis.

- **Local Variable.** The virtual callsite uses the value of a local variable in stack, whose value is loaded by a previous virtual call, as the function argument. In

Algorithm 7 Detect Argument Expansion Gadget

```
1: Input: f: a candidate virtual function
2: Output: determine whether f is an argument expansion gadget
3: args = getFunctionArgument(f)
4: paths = BFS(f)
5: for each path p in paths do
6:     ctrl_set = ∅
7:     ctrl_set = ctrl_set ∪ {rdi}  // Linux
8:     for each instruction inst in p do
9:         if isVirtualCall(inst) then
10:            regs = getArgumentRegs(inst)
11:            if regs ⊆ ctrl_set and |regs| > |args| then
12:                return true
13:            end if
14:        end if
15:        updateCtrlSet(inst, ctrl_set)
16:    end for
17: end for
18: return false
```

our analysis, we add a local variable into $ctrl\_set$ if there exists an instruction that loads the address of a local variable into argument register before a virtual callsite. At the next virtual callsite, we check whether its argument registers are in $ctrl\_set$ or not.

In the case of return value, the previous function call can be seen as a memory-read function. In the case of local variable, the previous function can be seen as a memory-write function. The searching for these functions is the same as in COOP, and we omit the details here. We implement a tool to collect such simple gadgets for future use during exploitation. Notice that, in our evaluation, we can always find such memory-read and memory-write gadgets, and we will present details of such gadgets we used in our PoC exploits in Section 5.5.4.

In our searching for argument expansion gadgets, we focus on two types of instruc-

tions, from the categories proposed in [109] and [113], to track the data flow. For a register passing instruction (e.g., `mov rcx,rdx`), if the source register (e.g., `rdx`) belongs to *ctrl_set*, the target register (e.g., `rcx`) will be added to *ctrl_set*; otherwise, the target register will be removed from *ctrl_set*. Similarly, for a memory dereference instruction (e.g., `mov rcx,[rdx+0x28]`), we will add the target register (e.g., `rcx`) to *ctrl_set* if the source register (e.g., `rdx`) belongs to *ctrl_set*, and remove the target register from *ctrl_set* otherwise.

Further, if there exists a direct function call in the candidate virtual function, we take a different analysis strategy on the direct function based on its size. If its number of basic blocks is less than 10 (empirically established as a good value), we continue our forward analysis; otherwise, we assume that values in the `rax`, `rbx`, `rcx`, `rdx` registers are all corrupted, and stop our analysis in this function. Although continuing our forward analysis regardless of the size of a function is possible, it will introduce unacceptable performance overhead in our searching and add more complexity in exploit generation. For the same reason, our forward analysis will skip the virtual functions that have more than 40 basic blocks.

5.4.3 Discovery of Transfer Gadgets

To search for transfer gadgets, we adopt four searching conditions. **TC1**: the function is a virtual function; **TC2**: the function has less than 10 instructions and its CFG contains only one basic block; **TC3**: the function ends with an indirect jump and there is no other virtual callsite in the function; and **TC4**: the `this` pointer used to invoke the function is different from the `this` pointer used at the virtual callsite in the function. By **TC2** and **TC3**, we reduce the complexity of transfer gadgets for the ease of exploit generation; i.e., the shorter and simpler is the gadget, the less complexity it brings.

For each candidate virtual function identified by Algorithm 6 (i.e., satisfying **TC1**), we first check whether it satisfies **TC2**, **TC3** and **TC4**. If yes, we then

apply backward analysis from the virtual callsite to find the memory dereference instruction that fetches the function pointer from the vtable and extract the dispatch offset. Recall that, during the identification of virtual functions, we also compute the vtable offset.

Given the vtable offset and dispatch offset of transfer gadgets, the searching for a chain of transfer gadgets to divert the control flow from a vulnerable callsite to a target function can be transformed to a graph searching problem. Given a directed graph $G = (V, E)$, V contains a vertex v_i if i is a dispatch offset or vtable offset of a transfer gadget, and an edge (v_i, v_j) belongs to E if there exists a transfer gadget that adjusts the dispatch offset from i to j. If the dispatch offset used at the vulnerable callsite is o_1 and the target function's vtable offset is o_2, our goal is to detect if there exists a path from v_{o_1} to v_{o_2}. Therefore, we use breadth-first-search (BFS) starting from v_{o_1} in the constructed graph to mark every visited vertex and check whether v_{o_2} is marked in the end.

5.4.4 Discovery of Invoking Gadgets

For searching invoking gadgets in the binary without source code, we choose two conditions. **IC1**: the function pointer in a candidate gadget is fetched via memory dereferencing only once; and **IC2**: the function pointer in a candidate gadget is independent from the `this` pointer at virtual callsite. A virtual function gadget will be taken as an invoking gadget, if one of the two conditions is satisfied. **IC1** comes from the fact that a successful virtual function call requires at least two memory dereferencing operations, i.e., vtable pointer fetch and virtual function pointer fetch. **IC2** follows the same rule in [108], i.e., the function pointer is not fetched via `this` pointer at callsite.

To search gadgets satisfying **IC1**, we apply backward analysis from the indirect callsite. The analysis will not stop until it reaches the entry point of the function or another function call. To search gadgets satisfying **IC2**, we pick the gadget if the

function pointer is fetched via a global variable or an other argument register (except for `this` pointer).

For searching invoking gadgets in the binary with source code available, we leverage one more condition in our search. **IC3**: the candidate virtual function gadget takes exactly four argument registers, and contains less than four basic blocks and less than 20 instructions. We check the searched candidate gadget satisfying **IC3** against the source code and choose the function if the indirect call in the function body is not a virtual function call. The restriction on size is to reduce the checking overhead.

Notice that the invoking gadget is not the contribution of our paper, but to ensure the completeness of this paper.

5.4.5 Tool Implementation

We implemented our detection tool with Python script using IDA 6.5. As TypeArmor is designed to protect binaries under 64-bit platforms, our tool targets binaries on X64 platforms. To search for candidate virtual functions in a target binary, we use the internal function `XrefTo` provided by IDAPython [20] to check whether the address of a function appears in a read-only region. We also use this function to identify the top of a vtable by checking whether the address is referenced in a function. In our BFS for the chain of transfer gadgets, we set the maximum length of the chain to be 10 to reduce the searching time. Note that, in our evaluation, the longest gadget chain we have seen contains 6 gadgets, which diverts control flow from vulnerable callsite to argument expansion gadget in Internet Explorer 10.

5.5 Evaluation

We evaluate the effectiveness of our LOOP attacks on both Linux and Windows X64 platforms. Since TypeArmor is only designed to protect 64-bit applications, we will

```
void ExecutableAllocator::reprotectRegion(void* start, size_t size,
    ProtectionSetting setting) {
        startPtr = reinterpret_cast<intptr_t>(start);
        mprotect(pageStart, size, (setting == Writable) ? RW : RX);
}
```

Listing 5.4: Wrapper Function ExecutableAllocator::reprotectRegion in Firefox

not discuss the LOOP attacks on X86 platforms. However, LOOP also works on X86 platforms, because LOOP relies on explicit data flow to set argument values, which is independent from platforms.

In particular, we launch LOOP attacks on Firefox on Linux platform as well as Flash Player and Internet Explorer on Windows platform. First, we show the availability of argument expansion gadgets and invoking gadgets to invoke a system call to change the memory protection flag without being blocked by TypeArmor (Section 5.5.1, 5.5.2 and 5.5.3). Then we show the availability of transfer gadgets to bypass both TypeArmor and vfGuard simultaneously, and extract part of the gadgets in gadget chain to give a better view of one full exploit (Section5.5.4). Finally we evaluate the availability and complexity of available gadgets in some common software or libraries (Section 5.5.5).

5.5.1 LOOP Attack on Firefox on Linux

Our LOOP attack calls *mprotect* on Linux platform through the chaining of argument expansion gadgets found in *libxul.so* and *libicui18n.so* in Firefox 46.0.

Argument Expansion Gadgets. We pick two argument expansion gadgets to expand the argument register count from 1 to 4. The first gadget expands the argument register count from 1 to 3, and the second gadget expands the argument register count from 2 to 4. Hijacking the control flow to a gadget with fewer function arguments will not be blocked by TypeArmor (see Section 5.3.3). Note that both gadgets use the return value of a previous call to set argument registers, and we select two functions to read memory, as discussed in Section 5.4.2.

Function Call to *mprotect*. We use an invoking gadget satisfying **IC3** (i.e., Listing 5.3) in *libicui18n.so*. As sensitive functions can only be called through direct calls in some mitigation (e.g., [134]), we use a wrapper function in *libxul.so*, which is invoked through the indirect call in Listing 5.3, to call *mprotect* (Listing 5.4). Here we still expand the argument register count from 1 to 4 to demonstrate the effectiveness of our tool while achieving our attacking goal.

Proof-of-Concept Exploit. We take CVE-2016-9079 [1], a use-after-free vulnerability, to build a PoC exploit and demonstrate our LOOP attack on Firefox 46.0. In our PoC exploit, we trigger a race condition to overwrite the metadata in an Array object to gain a stable capability of arbitrary reading and writing. We modify the public exploit on 32-bit Windows platform [10] to make it work on 64-bit Linux platform.

5.5.2 LOOP Attack on Flash Player on Windows

To evaluate our LOOP attack on Windows platform, we apply the LOOP attack to Adobe Flash Player 18.0.0.160. In particular, we target the OLE control extension of Flash Player 18.0.0.160 loaded by Internet Explorer, i.e., using the gadgets in *Flash64_18_0_0_160.ocx*, *ole32.dll* and *oleaut32.dll*.

Argument Expansion Gadgets. We choose two argument expansion gadgets to expand the argument register count from 1 to 4. They respectively expand the argument register count from 1 to 2 and from 2 to 4.

Function Call to *VirtualProtect*. On Flash Player, we pick *sub_307F56CC* in Listing 5.5 as the invoking gadget to call *VirtualProtect*. This function pointer in the gadget is fetched via one memory dereference operation and the gadget satisfies **IC1**.

Proof-of-Concept Exploit. To construct a PoC exploit, we take CVE-2015-5119 [8], which is an use-after-free vulnerability on Flash 18.0.0.160, to demonstrate that our LOOP attack also works on Windows X64 platform. To gain the ability to

```
1 Flash!sub_307F56CC
2    sub        rsp, 28h
3    mov        rax, [rcx+0x28]
4    test       rax, rax
5    jz          loc_307F56DF
6    mov        r8, [rcx+0x30]
7    call       rax
8 loc_307F56DF:
9    add        rsp, 28h
10   retn
```

Listing 5.5: Invoking Gadget in Flash Player of IC1

```
1  Ptls6::CLsBlockObject::Display
2  sub       rsp, 38h
3  mov       rax, [rcx+10h]
4  movups    xmm0, xmmword ptr [rcx+18h]
5  mov       r8, rdx
6  mov       r9, [rax+8]
7  lea       rdx, [rsp+38h+var_18]
8  mov       rcx, [r9+8]
9  movdqu    [rsp+38h+var_18], xmm0
10 call      qword ptr [r9+30h]
11 add       rsp, 38h
12 retn
```

Listing 5.6: Invoking Gadget in IE of IC2

arbitrarily read and write under 64-bit systems, we partially modify the exploit in [11] to achieve this in a 64-bit process by corrupting metadata in a ByteArray object.

5.5.3 LOOP Attack on Internet Explorer on Windows

To evaluate our LOOP attack on Windows platform, we also attack Internet Explorer 10 (10.00.9200.16438), using the gadgets found in the dynamic libraries *mshtml.dll* and *ieframe.dll*.

Argument Expansion Gadgets. We choose one gadget in *ieframe.dll* to expand the argument register count from 1 to 3; and we choose one gadget in *mshtml.dll* to expand the argument register count from 3 to 4.

Function Call to *VirtualProtect*. We pick *Ptls6::CLsBlock-Object::Display* as the invoking gadget, which satisfies **IC2**. However, we can hardly find a wrapper function to call *VirtualProtect* in *mshtml.dll* and *ieframe.dll*. Therefore, we load *Flash64_18_0_0_160.ocx*, and invoke *VirtualProtect*, using the same wrapper function as in Section 5.5.2, to change the protection flag of the desired memory in our exploit.

Proof-of-Concept Exploit. To demonstrate our LOOP attack on Internet Explorer 10, we take CVE-2013-2551 [25], an integer overflow vulnerability which can lead to arbitrary code execution. We implemented our PoC exploit on Internet Explorer 10 on Window 7 platform to change the protection flag.

5.5.4 LOOP Attacks Against vfGuard and TypeArmor

In Section 5.5.1, 5.5.2 and 5.5.3, we briefly introduce our exploit chain, including necessary argument expansion gadgets, invoking gadgets, and the CVEs we use to corrupt the memory. Now we use identified transfer gadgets to chain argument expansion gadgets together to bypass TypeArmor and vfGuard simultaneously to demonstrate the strength of LOOP attacks. For Firefox, we select nine transfer gadgets and two argument expansion gadgets to expand argument from 1 to 4. For Flash Player, we pick nine transfer gadgets and two argument expansion gadgets to expand argument from 1 to 4. For Internet Explorer, we choose thirteen transfer gadgets and two argument expansion gadgets to expand argument from 1 to 4.

Table 5.1 list the used gadgets in our LOOP attacks on Firefox. Gadget lists of Flash Player and Internet Explorer are listed in , Table A.1 and A.2. In the *Type* column, we list the type of the gadgets. *T* denotes transfer gadget, *AE* denotes argument expansion gadget, and *I* denotes invoking gadget. The *Vtable Offset* column reports the vtable offset of a gadget, and the *Dispatch Offset* reports the dispatch offset used at the virtual callsite in a gadget. The *Argument* column shows the expansion of argument register count.

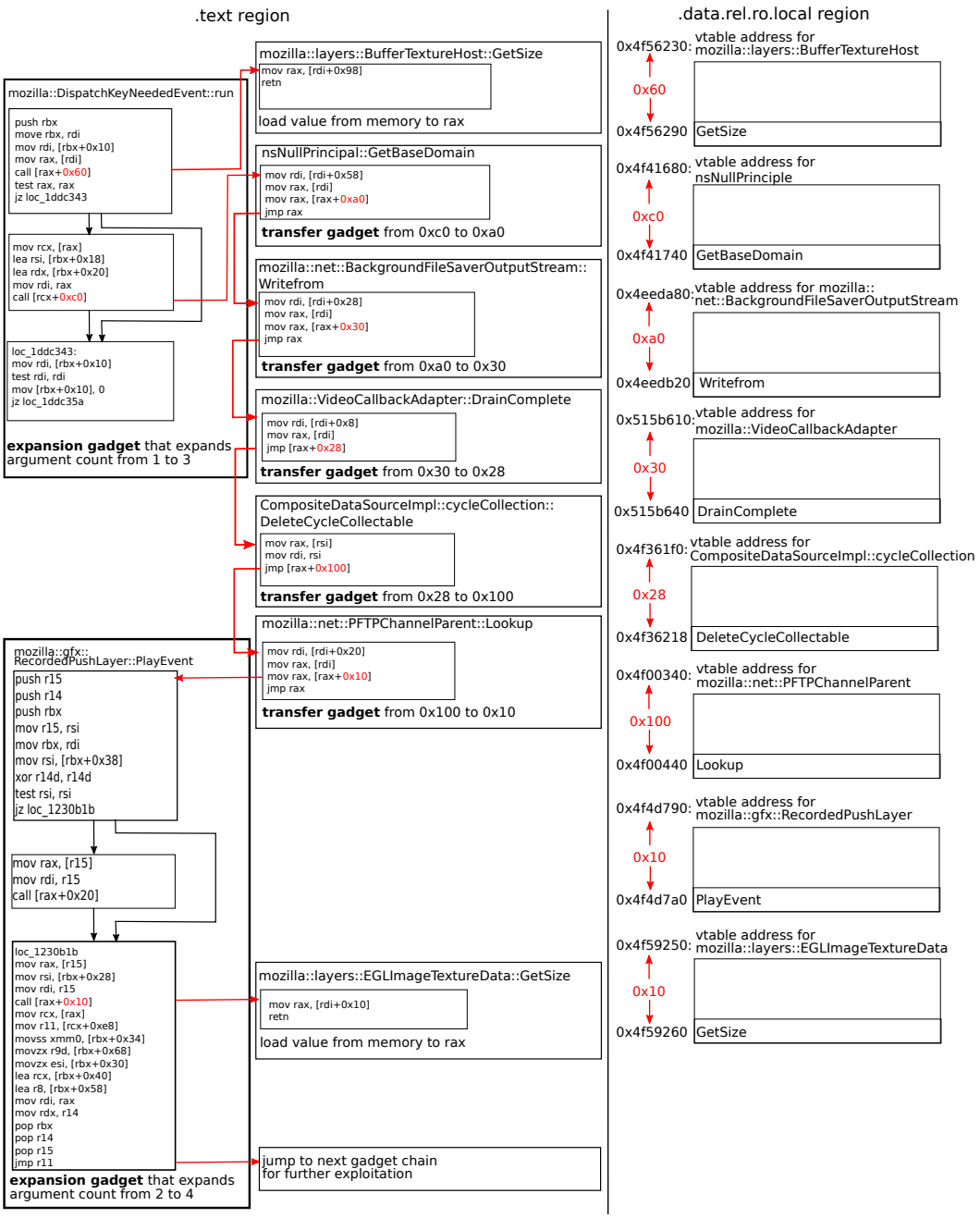


Figure 5.4: Part of Chained Gadgets Used in Our Firefox Exploit on Linux. The Red Line Denotes the Indirect Control Transfer During the Exploit. The Left-Side Lists all the Involved Virtual Functions in *.text* Region. The Right-Side Lists all the vtables that the Virtual Functions on the Left-Side Belong to.

Table 5.1: The Execution Flow of the LOOP Attack on Firefox

Step	Gadget Name	Type	VTable Offset	Dispatch Offset	Argument
1	mozilla::plugins::PPluginInstanceParent:: AdoptSharedMemory	T	0x298	0x30	–
2	DigestOutputStream::Close	T	0x30	0x18	–
3	DispatchKeyNeededEvent::Run	AE	0x18	0xC0	1 to 3
4	nsNullPrinciple::GetBaseDomain	T	0xC0	0xA0	–
5	BackgroundFileSaverOutputStream:: WriteFrom	T	0xA0	0x30	–
6	VedeoCallbackAdapter::DrainComplete	T	0x30	0x28	–
7	cycleCollection::DeleteCycleCollectable	T	0x28	0x100	–
8	FTPChannelParent::Lookup	T	0x100	0x10	–
9	RecordedPushLayer::PlayEvent	AE	0x10	0xE8	2 to 4
10	PFilePickerChild:: CreateSharedMemory	T	0xE8	0x28	–
11	FakeSynthCallback::cycleCollection:: FDeleteCycleCollectable	T	0x28	0x38	–
12	ReplaceableGlue::Copy	I	0x38	–	–

Moreover, Fig. 5.4 shows part of the gadget chain of our PoC exploit on Firefox (i.e., Step 3–9 in Table 5.1). The two argument expansion gadgets are *mozzilla::DispatchedKeyNeededEvent::run* and *mozilla::gfx::RecordedPushLayer–::playevent*, shown in the leftmost of Fig. 5.4. With forward analysis on *mozzilla::-DispatchedKeyNeededEvent::run*, we can derive that only one argument register is marked as R. At the virtual callsite `call [rcx+0xc0]`, argument registers `rsi` and `rdx` are loaded with value under attacker's control. Thus, this argument expansion gadget expands argument register count from 1 to 3. Similarly, *mozilla::gfx::RecordedPushLayer::playevent* expands argument register count from 2 to 4 (here we omit the value loaded into `r8` and `r9` as in our PoC exploit, we only need to control the first four argument registers to invoke *mprotect*).

However, vfGuard can block the diverted control from the first argument expan-

Table 5.2: The Number of Argument Expansion Gadgets Discovered in Some Common Software or Libraries.

Module Name	1 to 2	1 to 3	1 to 4	2 to 3	2 to 4	3 to 4
libxul.so (46.0)	11	5	2	3	6	1
libicui18n.so(5.6)	0	1	0	0	0	0
Flash64_18_0_0_160.ocx (18.0.0.160)	6	2	1	7	5	0
ole32.dll (6.1.7601.17514)	2	0	0	1	1	1
oleaut32.dll (6.1.7601.17514)	0	0	0	0	0	0
mshtml.dll (10.00.9200.16438)	13	5	4	16	4	4
ieframe.dll (10.00.9200.16438)	0	0	1	6	1	1
chrome.dll (56.0.2924.87)	79	53	45	75	55	19
libQt5WebKit.so (5.1.1)	0	0	4	9	7	0

sion gadget to the second one. Thus, we use five transfer gadgets (shown in the middle of Fig. 5.4) to chain them together. These transfer gadgets exist in *libxul.so*; and all the involved vtables are also listed in the rightmost of Fig. 5.4, including the top address of a vtable and the address where the function pointer resides in.

5.5.5 Availability and Complexity of Gadgets

We also evaluate the availability and complexity of gadgets used in our LOOP attack for Firefox, Flash Player and Internet Explorer as well as in some libraries of Chrome and Qt 5 (i.e., *chrome.dll* on Windows and *libQt5WebKit.so* on Linux. We use the number of gadgets to measure availability and the average number of basic blocks/instructions to measure complexity.

Table 5.2 and Table 5.3 reports the availability and complexity of argument expansion gadgets. We can see from the second to the seventh columns that argument expansion gadgets with different expansion capability are available in most modules, which indicates that these gadgets are sufficient to construct our LOOP attacks. With respect to the complexity, one gadget contains 1.2 basic blocks and 7.9 instruc-

Table 5.3: The Complexity of Argument Expansion Gadgets Discovered in Some Common Software or Libraries.

Module Name	Avg. Block	Avg. Inst.	Member Var.	Return Value	Local Var.
libxul.so (46.0)	1.72	14.96	19	9	0
libicui18n.so(5.6)	1.00	7.00	1	0	0
Flash64_18_0_0_160.ocx (18.0.0.160)	1.10	9.43	16	5	0
ole32.dll (6.1.7601.17514)	1.00	5.00	5	0	0
oleaut32.dll (6.1.7601.17514)	–	–	–	–	–
mshtml.dll (10.00.9200.16438)	1.63	9.21	41	4	1
ieframe.dll (10.00.9200.16438)	1.00	5.33	9	0	0
chrome.dll (56.0.2924.87)	1.12	6.45	310	16	0
libQt5WebKit.so (5.1.1)	1.00	6.10	20	0	0

tions on average. Besides, we also report the type of identified argument expansion gadget in the last three columns. The most common type is *Member Variable*, which widely exists in most modules. *Return Value* only exists in large and primary modules that implement the most functionalities, i.e., *libxul.so*, *Flash.ocx*, *mshtml.dll* and *chrome.dll*. In *mshtml.dll*, we also find one gadget of type *Local Variable* which contains 8 basic blocks and 66 instructions.

Table 5.4 reports the availability and complexity of transfer and invoking gadgets. For transfer gadgets, we report the total number of discovered transfer gadgets as well as the number of transfer gadgets that contain distinct pair of dispatch offset and vtable offset in the second and third columns respectively. The results indicate that transfer gadgets are quite common in all modules. Besides, each transfer gadget contains one basic block and 4.3 instructions on average.

For invoking gadgets, we list the number of different types of invoking gadgets from the first to the third columns of Table 5.5. As some gadgets may fall in **IC1** and **IC2** at the same time, we list the invoking gadgets that satisfy both conditions in the parentheses under column *IC1* and column *IC2* reports the number of invoking

Table 5.4: The Statistics of Transfer Gadgets Discovered in Some Common Software or Libraries

Module Name	Total	Distinct	Avg. Block	Avg. Inst.
libxul.so (46.0)	6692	1007	1	4.24
libicui18n.so(5.6)	32	20	1	7.03
Flash64_18_0_0_160.ocx (18.0.0.160)	273	133	1	4.00
ole32.dll (6.1.7601.17514)	137	19	1	3.35
oleaut32.dll (6.1.7601.17514)	47	7	1	3.40
mshtml.dll (10.00.9200.16438)	639	105	1	3.60
ieframe.dll (10.00.9200.16438)	193	55	1	3.37
chrome.dll (56.0.2924.87)	4529	510	1	4.72
libQt5WebKit.so (5.1.1)	508	107	1	5.09

gadget that only satisfies **IC2**. Except for **IC3**, invoking gadgets satisfying **IC1** and **IC2** are all quite common in most modules. Besides, on average, each invoking gadget contains 1.5 basic blocks and 12.6 instructions.

Due to the low complexity of argument expansion gadgets, transfer gadgets and invoking gadgets, there is a low chance for them to have side effect. Specifically, we manually analyse the argument expansion gadgets in *libxul.so*, where the maximum number of basic blocks and instructions is respectively 5 and 25; and they all have no side effect. However, it does not mean that we cover all feasible gadgets in binary. Dedicated attackers may seek more complicated gadgets to launch the attack. It is also worth mentioning that the gadgets in Section 5.5.1, 5.5.2, 5.5.3 and 5.5.4 are all taken from Table 5.2, Table 5.4 and Table 5.5.

Table 5.5: The Statistics of Invoking Gadgets Discovered in Some Common Software or Libraries

Module Name	IC1	IC2	IC3	Avg. Block	Avg. Inst.
libxul.so (46.0)	334 (44)	229	6	1.72	19.45
libicui18n.so(5.6)	0 (0)	0	1	1.00	8.00
Flash64_18_0_0_160.ocx (18.0.0.160)	27 (12)	44	0	1.46	17.48
ole32.dll (6.1.7601.17514)	4 (1)	1	0	2.20	6.60
oleaut32.dll (6.1.7601.17514)	3 (0)	1	0	1.75	10.75
mshtml.dll (10.00.9200.16438)	22 (15)	4	0	1.19	14.88
ieframe.dll (10.00.9200.16438)	13 (3)	2	0	1.67	7.13
chrome.dll (56.0.2924.87)	1741 (451)	135	0	1.51	15.89
libQt5WebKit.so (5.1.1)	41 (5)	32	0	1.37	12.99

5.6 Discussion

5.6.1 Non-Void Function Overesitimation

TypeArmor also blocks the control flow transfer from a non-void callsite to a void function. To judge whether a callee function is void or non-void, TypeArmor applies backward analysis starting from the exit points of the function to search for write operations on `rax` that may indicate the setting of return value. If such operations exist, the function will be identified as non-void; otherwise, it will be identified as void.

Under this enforcement strategy, TypeArmor blocks the control flow transfer in Control Jujutsu attack [74]. The attacker in Control Jujutsu diverts the control flow from a non-void callsite in *ng_output_chain* to a void function *ngx_execute_proc*. The callee function *ngx_execute_proc* calls *execve*, and returns with no write operation on `rax`. In this case, TypeArmor can correctly identify that *ngx_execute_proc* is a void function and block the control flow transfer. However, the situation is different for virtual function gadgets in LOOP.

Table 5.6: COOP and LOOP against binary-level mitigation

Mitigation Type	Mitigation Technique	COOP	LOOP
Code Diversification	STIR [125]	✗	✗
VTable Integrity Enforcement	T-VIP [76]	✗	✗
	VTint [133]	✗	✗
Generic CFI	CFI [40]	✗	✗
	O-CFI [93]	✗	✗
	CCFIR [134]	✗	✗
	LockDown [102]	✗	✗
Semantic-Based CFI	vfGuard [103]	✓	✗
	TypeArmor [124]	✓	✗

As TypeArmor does not resolve the target function at indirect call/jump, it cannot judge whether the target function sets the value in `rax` or not. For transfer gadgets that end with an indirect call/jump, backward analysis to search for write operation on `rax` does not work. Hence, TypeArmor cannot decide whether a transfer gadget is a void function or not, and has to regard a transfer gadget as a non-void gadget for compatibility issue. In addition, the correctness to identify void functions in binary in TypeArmor is low (17.89%) [124]. We believe that the existence of transfer gadgets is one main reason for such a low correctness. In summary, the enforcement strategy that a non-void callsite cannot target a void function is also not effective in blocking control flow transfers in LOOP.

5.6.2 Implication on Binary-Level Mitigation

We assess the existing binary-level mitigation against LOOP attacks, and also compare LOOP attacks with COOP attacks. Table 5.6 reports the results, where the first two columns list the mitigation type and the specific mitigation technique.

STIR [125] prevents attackers from accessing gadgets for ROP chain construction via reordering the basic blocks of all functions at binary level. However, such a code diversification defense is only effective against ROP gadgets with a short sequence of instructions. In LOOP attacks, the attacker uses the whole virtual function as one gadget and loads value into argument register via explicit data flow. Thus, STIR is not effective against vtable reuse attacks.

VTint [133] is a vtable integrity enforcement at binary level. At a virtual callsite, it instruments checks to verify whether the vtable is located in a read-only region. Similarly, T-VIP [76] attempts to move all the identified vtables into a new read-only region and instrument at each virtual callsite to check whether the vtable pointer points to this region. Since our LOOP attack uses the existing vtable in the read-only area, both T-VIP and VTint cannot mitigate our LOOP attack.

O-CFI [93] is a coarse-grained CFI [40] at the binary level. It combines a similar strategy of STIR to randomize the code layout for eliminating available gadgets to attacker. However, as previously discussed, such an information hiding strategy cannot prevent attackers from gaining the knowledge of vtable location and loading desired value into argument register. Therefore, O-CFI is unable to block vtable reuse attacks.

CCFIR [134] is also a binary-level coarse-grained CFI, where an indirect call/jump can only jump to an address-taken function pointer and sensitive functions (e.g., *mprotect*) can only be invoked via direct calls. In LOOP, we craft the vtable pointer to divert the control flow. To call *mprotect* or *VirtualProtect*, we use existing wrapper functions to call those sensitive functions. Therefore, CCFIR is unable to mitigate LOOP attacks.

LockDown [102] implements a modular CFI at binary level. At indirect callsite, only functions that are defined or imported in the current module are considered as valid targets. However, in our LOOP attack on Firefox, Flash Player and Internet Explorer, sensitive functions like *mprotect* and *VirtualProtect* exist in the Import

Address Table. These sensitive functions will be considered as valid targets under LockDown.

For TypeArmor [124] and vfGuard [103], we have shown that they are still vulnerable to our LOOP attacks. Compared to COOP attacks that can be blocked by TypeArmor and partially blocked by vfGuard, LOOP is more advanced.

5.6.3 Class Hierarchy

Class hierarchy information may be utilized to mitigate our LOOP attacks. Recently, Pawlowski et al. proposed Marx [101] to recover class hierarchy relation in C++ at binary level. They proposed a vtable protection policy based on the resolved class hierarchy. Then, Elsabagh et al. proposed VCI [72], combining Marx and vfGuard, to instrument before the identified virtual callsite to limit the valid targets. Both techniques try to recover the class hierarchy and layout relation from constructor functions. In VCI, for a virtual callsite where class type information is unresolved, it applies the same policy as in vfGuard.

The effectiveness of both mitigations against vtable reuse attack depends on how precisely they can resolve the class type information at virtual callsites. In particular, both mitigations perform backward data analysis from the target virtual callsite; and if there exists constructor function on its path, VCI is able to fully or partially resolve the class type information at virtual callsite. However, backward inter-procedure analysis becomes impractical for virtual functions. In LOOP attacks, argument expansion gadgets and transfer gadgets are all virtual functions that do not contain constructor function. Hence, the class type information at virtual callsites in those gadgets cannot be resolved; and VCI has to take the same enforcement policy as in vfGuard for the virtual callsites in those gadgets in LOOP. In addition, in the PoC exploits on Flash Player and Internet Explorer, we trigger LOOP attacks from *Object.toString* by overwriting the vtable pointer of *Object*. At the vulnerable callsite, we apply the same backward analysis as described in VCI, and verify that the type

information cannot be resolved.

In the evaluation of VCI on *libxul.so* [72], the class type information can be fully resolved for 32% of the identified virtual callsites. In the evaluation of Marx [101], the resolved class type information at the virtual callsites does not exceed 20% of the total number of virtual callsites. At present, such a low resolving rate of virtual callsite target is hard for us to evaluate their effectiveness against LOOP attacks. In future, we plan to explore how to increase the correct rate of resolving class type information and evaluate the effectiveness against LOOP.

5.7 Related Work

Non-Control Data Attacks. Non-control data attacks are proposed to corrupt the data that is irrelevant to control flow to build attacks. In Control Flow Bending [54], an attacker can corrupt the parameter in the *printf* function. In Data-Oriented Programming [82] and other non-control data attacks [58, 81], an attacker relies on non-security data to construct exploits. Data-Oriented Programming demonstrates the weakness in most modern defenses because data integrity enforcement is not in the protection scope of these defenses. In LOOP, we demonstrate the availability and feasibility of virtual function gadgets in complicated software to bypass the mitigation combining TypeArmor and vfGuard, two state-of-art defenses designed to enforce vtable integrity.

JOP And COP. In Jump-Oriented Programming (JOP) [50], control flow is hijacked via a dispatch table. In JOP the indirect call target can be any address in code region, but in LOOP the indirect call target can only be the entry point of a function. In Call-Oriented Programming (COP) [55], gadgets that end with indirect jump are combined via a ROP attack for a successful attack. Compared with COP, we propose different functional gadgets in LOOP, and present exploits under the protection of TypeArmor and vfGuard.

Newton. Newton [123] is a framework to find available gadgets for evaluating the feasibility of code reuse attacks with dynamic analysis. It presents PoC exploits aganst CPI [87] and PathArmor [122]. It is successfully applied to search for the gadgets in three open-source web servers (i.e., nginx, Apache, and lighttpd) to demonstrate the feasibility of Newton. CPI is a source-level based mitigation, which extracts the information of sensitive pointers and inserts integrity check in the compiled binary. PathArmor provides binary-level context-sensitive CFI on commodity hardware. However, it mainly focuses on C programs rather than C++ programs. Differently, we analyse the weakness in two binary-level mitigations (i.e., vfGuard and TypeArmor) and demonstrate that vtable reuse attacks are still feasible even if vfGuard and TypeArmor are deployed at the same time. We develop tools to find suitable gadgets in binary-code with static analysis and show PoC exploits on one open-source browser (Firefox) and two close-source software (Flash Player and Internet Explorer).

Information Hiding. Information hiding techniques prevent attackers from gaining useful information from target application and raise the bar for exploitation. The first category of work prevents attackers from gaining information. Oxymoron [46], Readactor [64] and Readactor++ [65] all replace code pointers in binary code with labels, and put those information in an area inaccessible from attackers. The second category of work prevents attackers from generating successful exploits. Heisenbyte [118] introduces destructive code read, where the binary code is destroyed after it is read. Similarly, NEAR (No-Execute-After-Read) scrambles the code after it is read using hypervisor [127]. TASR (Timely-Address-Space-Randomization) [47] rerandomizes the process memory layout at runtime whenever the process executes one sensitive output system call. CodeArmor [59] splits the binary code of a C program into concrete code space and virtual code space and hides the function pointer in concrete code. Moreover, it continues to randomize the address space of concrete code space to thwart diversification-aware code-reuse attacks. In our attacking

model, we assume that attackers can gain arbitrary read/write ability. Therefore, the information hiding techniques are not considered within our attacking model.

Code Diversification. Code diversification techniques make efforts to eliminate available gadgets in binary. XIFER [67] rewrites arbitrary binary and disperses fractions of binary code at the granularity of basic blocks. It rearranges the position of each basic block in functions and uses binary rewriter to keep CFG unchanged. G-Free [99] is a compiler-based approach to eliminate potential gadgets at the granularity of instructions. It rewrites instructions emitted into binary code to remove all potential byte sequences that could be used as gadgets. Isomemoron [68] loads one original module and one diversified module into memory space and performs the execution randomization at the granularity of function calls. At each function call at runtime, Isomemoron randomly continues the execution flow to the original one or the diversified one. The more complex the payload, the more secure the application. All these techniques aim to reduce the number of potential ROP gadgets. More importantly, all code rewriting techniques generate semantically equivalent code. As discussed in Section 5.6.2, code diversification techniques are not effective against our attack.

Hardware-Assisted CFI. Hardware-assisted CFI tries to enforce CFI via information retrieved from hardware. PathArmor [122] leverages LBR cache to provide context-sensitive CFI. It compares the paths stored in LBR with rightful order before invoking security-sensitive functions. It is designed to protect the indirect control flow in C programs. Hence, the virtual callsite in C++ applications is out of its protection scope. Similarly, kBouncer [100] and ROPecker [60] both use LBR cache to record the execution path of return instructions and prevent ROP attacks through some heuristics learned from ROP chains. However, these two mitigations have been demonstrated to be still vulnerable to ROP attacks in [55]. Furthermore, LOOP does not rely on constructing ROP chains for a successful attack and is out of their protection scope.

Binary-Level VTable Enforcement. VTGuard [92] is a vtable hijacking mit-

igation employed in Windows Internet Explorer, which inserts a secret cookie at the end of the vtable at compile-time and inserts check to verify the cookie of the vtable at runtime. However, VTGuard only protects partial virtual callsites in Internet Explorer. We check the binary code of the gadgets we use in our exploit on Internet Explorer 10, and find that virtual callsite in those gadgets are not protected by VTGuard. Therefore, VTGuard is unable to defend our vtable reuse attack. VTPin [107] is another binary-level mitigation against vtable hijacking attack. It prevents dangling pointer from dereferencing the injected fake vtable via use-after-free vulnerabilities. At the time of object deallocation, VTPin deallocates all space allocated, but preserves and replaces the vtable pointer of the object with a pointer referencing a pre-set safe vtable. In LOOP attacks, we do not rely on dangling pointers to launch vtable reuse attacks. Thus we believe VTPin is unable to defend our vtable reuse attacks.

Compiler-Based CFI. IFCC/VTV [119] enforces a fine-grained CFI on forward-edge control flow transfer. Modular CFI [96] implements a fine-grained CFI, which is thread-safe and support dynamic linking. Per-Input CFI [98] first calculates the full static CFG at compile time, then adds the activated edge into an enforced CFG at runtime and checks the validity of indirect jump/call target according to the enforced CFG. Cryptographically CFI [91] calculates the message authentication code of vtable pointers and object addresses to prevent vtable pointers from being corrupted by attackers. SafeDispatch [83] applies Class Hierarchy Analysis [71] (CHA) to determine the set of valid function pointers that may be invoked at virtual callsite for each object in the program. WIT [42] generates the set of objects that can be written by each instruction in the program by static analysis and prevents instructions from modifying objects that are not in the set. VTrust [131] summarizes a type list for each virtual function call at source code level and uses such information as the signature. CPI/CPS [87] tries to provide integrity enforcement on code pointers at a low performance overhead. In CPS, code pointers are moved to an isolated region inaccessible from attackers. In CPI, not only the code pointers but also the data

pointers that point to structures containing code pointers are moved to the isolated region. VTI [51] rearranges the memory layout of a vtable to improve the performance overhead of runtime check. CFIXX [53] proposes an orthogonal policy, Object Type Integrity (OTI) in its CFI implementation. CFIXX checks the object type at each virtual callsite and verifies if the object type matches the object type assigned in constructor function. These compiler-based mitigations accurately extract various kinds of information from source code and use them as authenticating message to verify indirect call target. However, things are complicated at binary level. As discussed in Section 5.6.3, the correct rate of resolving virtual call target is low. Binary-level mitigations have to make some conservative assumptions for compatibility issues. With LOOP, we demonstrate that those assumptions leave chances for attackers to construct successful exploits.

Other Mitigations. Data Flow Integrity (DFI) [56] is proposed to defend against non-control data attacks. Recently, Kenali [114] is proposed to enforce DFI in the Linux kernel for the 64-bit ARM architecture. DFI tries to verify that every value written into a variable comes from a legal source. For the reason of performance, both approaches require the knowledge of source code and generate a set of security-sensitive variables for further validation. Similar to information hiding techniques, DFI prevents attackers from reading/writing security-unrelated data, which violates the attacking model of LOOP.

Memory safety checks are effective in mitigating memory corruption. Cyclone [84] and CCured [95] introduce type-safe pointer in C. Cling [41] provides protection against use-after-free vulnerabilities. However, these checking schemes can only be applied to C programs, but not feasible for C++ programs. AddressSanitizer [110] enforces memory safety for C/C++ applications to detect memory corruptions. However, its high performance overhead and the increased usage of heap and stack make it impractical to run in real-world applications.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis presents some novel code reuse attacks against modern mitigation techniques. This section summarizes the contributions of this thesis.

6.1.1 Weakness in Constant Blinding PRNG

In Chapter 3, we demonstrate that the secret cookie for constant blinding in Flash Player is predictable due to its weak design and implementation of PRNG. We proposed two different methods to recover the seed value used in the PRNG. One is cryptanalysis based on the insufficient confusion and diffusion in hash function. Another one is the ill-considered implementation to store the seed value in memory. It enables attackers to find the seed value without much effort. Different from previous works that concentrate on some corner cases in constant blinding mechanism, we are the first to analyse the constant blinding PRNG and recover the seed value for defeating it.

6.1.2 Non-blinding Constant in Browser JIT engine

In Chapter 4, we propose Blockade, a grammar based fuzzing framework, to detect the non-blinded constant in JIT code. Besides the cases that have been discovered in previous work, our tool has identified more cases that have never been discussed before. More importantly, our framework could be easily extended to newly added grammar in JavaScript or other script language to detect potential non-blinded constant in future.

6.1.3 Layered Object Oriented Programming

In Chapter 5, we revisit the most advanced binary-level defenses (TypeArmor and vfGuard) against vtable reuse attacks and propose Layered Object-Oriented Programming (LOOP), a more sophisticated vtable reuse attack than COOP, to bypass TypeArmor and vfGuard. We propose the concept of argument expansion gadgets and transfer gadgets, and develop tools to find those gadgets. We also present some proof-of-concept exploits on Linux and Windows platform, which demonstrate that it is still possible to launch vtable reuse attacks in complicated software that TypeArmor and vfGuard intend to protect.

6.2 Future Work

According to the conclusions given in this thesis, we present some interesting and challenging directions in future work.

6.2.1 Advanced Exploitation Techniques on JIT compiler

Recently, more advanced JIT spray attacks were given in different attacking scenarios [77]. In the recent Pwn2Own hacking competition, JIT compiler has become a hot target for exploitation [61, 62, 117].

Current work shows that JIT spray attack is only part of exploitation techniques on JIT compiler. In future, JIT compiler will remain an attractive target in the field of security research. Any flaw in JIT compilers may bring unexpected attacking surfaces to hacking community.

6.2.2 Binary Level Mitigation

In the research on control flow enforcement during recent years, most of the works are based on source code and built on LLVM. Few works are built on binary. In this thesis, we demonstrate that two state-of-the-art mitigation techniques based on binary are still weak against code reuse attacks. A more fine grained enforcement policy should be designed in binary level security.

At present, Class Hierarchy Analysis (CHA) [71] seems a feasible solution. As discussion in Section 5.6.3, current success rate of resolving the indirect call target at virtual callsite is not high enough. It is therefore hard for us to evaluate their effectiveness against our attack. In future, more features should be extracted from binary code for enforcing security policy and increasing coverage.

6.2.3 Evaluation on Effectiveness of Mitigation

Following Section 6.2.2, we are lacking some formal methods and systematic approaches to evaluate the effectiveness of a proposed mitigation technique. At present, a common practice to evaluate a mitigation technique is to apply the mitigation in software against known exploits in the wild. However, such limited evaluation cannot guarantee the security against variants of known attacking techniques, like LOOP.

One possible direction is Automatic Exploit Generation (AEG) [45] technique, which aims to generate a working exploit automatically. Such technique can greatly help to assess the exploitability of a memory corruption error in the target application. In the meantime, AEG can also help evaluate the effectiveness of some mitigation techniques via judging whether working exploit exists [81,129] or some specific condi-

tions are satisfied [80]. However, it requires quite a lot of effort from both attacking side and defending side to cover every possible corner cases in different exploitation techniques.

Bibliography

[1] Bugzilla CVE-2016-9079. https://bugzilla.mozilla.org/show_bug.cgi?id=1321066. Last Access: 2018-07-19.

[2] Bypass DEP and CFG Using JIT Compiler in Chakra Engine. http://xlab.tencent.com/en/2015/12/09/bypass-dep-and-cfg-using-jit-compiler-in-chakra-engine/. Last Access: 2018-07-19.

[3] Compiler-Based Security Mitigations in Android P. https://android-developers.googleblog.com/2018/06/compiler-based-security-mitigations-in.html. Last Access: 2018-08-03.

[4] Data Execution Prevention. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx. Accessed: 2018-06-07.

[5] Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457155(v=technet.10). Last Access: 2018-07-19.

[6] DirtyCow. https://dirtycow.ninja. Last Access: 2018-08-03.

[7] ECMAScript 2016 Language Specification. https://www.ecma-international.org/ecma-262/7.0/. Last Access: 2018-07-19.

[8] Exploit-DB CVE-2015-5119. https://www.exploit-db.com/exploits/37523/. Last Access: 2018-07-19.

[9] Exploit-DB CVE-2015-5122. https://www.exploit-db.com/exploits/37599/. Last Access: 2018-06-07.

[10] Exploit-DB CVE-2016-9079. https://www.exploit-db.com/exploits/41151/. Last Access: 2018-07-19.

[11] Exploiting a 64-bit Browser with Flash CVE-2015-5119. https://community.rapid7.com/community/metasploit/blog/2015/07/31/supporting-a-64-bits-renderer-on-flash-cve-2015-5119. Last Access: 2018-07-19.

[12] Explore Terms: A Glossary of Common Cybersecurity Terminology. https://niccs.us-cert.gov/glossary. Last Access: 2018-08-04.

[13] Fireeye Threat Research on CVE-2015-5122. https://community.fireeye.com/external/1311. Last Access: 2018-08-03.

[14] Forbes: Shopping for Zero-Days: A Price List for Hackers' Secret Software Exploits. https://www.forbes.com/sites/andygreenberg/2012/03/23/shopping-for-zero-days-an-price-list-for-hackers-secret-software-exploits. Last Access: 2018-08-03.

[15] From Browser To System Compromise. https://www.blackhat.com/docs/us-16/materials/us-16-Molinyawe-Shell-On-Earth-From-Browser-To-System-Compromise-wp.pdf. Last Access: 2018-08-04.

[16] GCC 4.1 Release Series Changes, New Features, and Fixes. https://gcc.gnu.org/gcc-4.1/changes.html. Last Access: 2018-08-04.

[17] /GS (buffer security check). https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check. Last Access: 2018-08-04.

[18] HackerOne. https://www.hackerone.com/. Last Access: 2018-08-03.

[19] HeartBleed Vulnerability. http://heartbleed.com. Last Access: 2018-08-03.

[20] IDAPython. https://www.hex-rays.com/products/ida/support/idapython_docs. Last Access: 2018-08-02.

[21] iDefence Lab. https://labs.idefense.com/. Last Access: 2018-08-03.

[22] Inside AVM. https://recon.cx/2012/schedule/attachments/43_Inside_AVM_REcon2012.pdf. Last Access: 2018-06-07.

[23] JavaScript 2.0 Grammar. http://www-archive.mozilla.org/js/language/js20-1999-02-18/grammar.html. Last Access: 2018-07-19.

[24] Microsoft: Control Flow Guard. https://docs.microsoft.com/en-us/windows/desktop/secbp/control-flow-guard. Last Access: 2018-08-03.

[25] N. Joly. Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013). http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php. Last Access: 2015-05-21.

[26] NVD CVE-2017-0143. https://nvd.nist.gov/vuln/detail/CVE-2017-0143. Last Access: 2018-08-03.

[27] NVD CVE-2017-0144. https://nvd.nist.gov/vuln/detail/CVE-2017-0144. Last Access: 2018-08-03.

[28] NVD CVE-2017-0145. https://nvd.nist.gov/vuln/detail/CVE-2017-0145. Last Access: 2018-08-03.

[29] NVD CVE-2017-0146. https://nvd.nist.gov/vuln/detail/CVE-2017-0146. Last Access: 2018-08-03.

[30] NVD CVE-2017-0147. https://nvd.nist.gov/vuln/detail/CVE-2017-0147. Last Access: 2018-08-03.

[31] PaX Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt. Last Access: 2018-06-07.

[32] PyKd - Python Extension for WinDBG to Access Debug Engine. https://archive.codeplex.com/?p=pykd. Last Access: 2018-08-03.

[33] /SAFESEH (image has safe exception handlers). https://msdn.microsoft.com/en-us/library/9a89h429.aspx. Last Access: 2018-08-04.

[34] System V Application Binary Interface. https://www.uclibc.org/docs/psABI-x86_64.pdf. Last Access: 2018-07-19.

[35] Trend Micro Presenting Mobile Pwn2Own 2016. https://blog.trendmicro.com/presenting-mobile-pwn2own-2016. Last Access: 2018-08-03.

[36] Trend Micro Pwn2Own The Root of Research. https://blog.trendmicro.com/pwn2own-the-root-of-research/. Last Access: 2018-08-03.

[37] Windows Parameter Passing. https://msdn.microsoft.com/en-us/library/zthk2dkh.aspx. Last Access: 2018-07-19.

[38] Windows Software Security Defense. https://msdn.microsoft.com/en-us/library/bb430720.aspx. Accessed: 2018-06-07.

[39] ZDI: Pwn2Own 2018 Rules. https://www.zerodayinitiative.com/Pwn2Own2018Rules.html. Last Access: 2018-08-03.

[40] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control Flow Integrity. In *12th ACM Conference on Computer and Communications Security (CCS)*, pages 340–353. ACM, 2005.

[41] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *19th USENIX Security Symposium (USENIX Security)*, pages 177–192. USENIX Association, 2010.

[42] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *29th IEEE Symposium on Security and Privacy (S&P)*, pages 263–277. IEEE, 2008.

[43] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L Schuff, David Sehr, Cliff L Biffle, and Bennet Yee. Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code. In *ACM SIGPLAN Notices*, volume 46, pages 355–366. ACM, 2011.

[44] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *13th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[45] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic Exploit Generation. *Communications of the ACM*, 57(2):74–84, 2014.

[46] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *23rd USENIX Security Symposium (USENIX Security)*, pages 433–447. USENIX Association, 2014.

[47] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *22nd ACM Conference on Computer and Communications Security (CCS)*, pages 268–279. ACM, 2015.

[48] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy (S&P)*, pages 227–242. IEEE, 2014.

[49] Dion Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. *BlackHat DC*, 2010.

[50] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: a New Class of Code-Reuse Attack. In *6th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, pages 30–40, 2011.

[51] Dimitar Bounov, Rami Kici, and Sorin Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *14th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2016.

[52] John E Bryson. NIST Special Publication 800-53 Revision 4 Security and Privacy Controls for Federal Information Systems and Organizations JOINT TASK FORCE TRANSFORMATION INITIATIVE. 2012.

[53] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++. In *16th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2018.

[54] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Security)*, pages 28–38. USENIX Association, 2015.

[55] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security)*, pages 385–399. USENIX Association, 2014.

[56] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–160. USENIX Association, 2006.

[57] Ping Chen, Rui Wu, and Bing Mao. JITSafe: a Framework against Just-In-Time Spraying Attacks. *IET Information Security*, 7(4):283–292, 2013.

[58] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-Control-Data Attacks Are Realistic Threats. In *14th USENIX Security Symposium (USENIX Security) - Volume 14*, SSYM'05, pages 12–12. USENIX Association, 2005.

[59] Xi Chen, Herbert Bos, and Cristiano Giuffrida. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 514–529. IEEE, 2017.

[60] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attack. In *12th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[61] Dustin Childs. The Results - Pwn2Own 2017 Day One. https://blog.trendmicro.com/resultspwn2own-2017-day-one/. Last Access: 2018-06-07.

[62] Dustin Childs. The Results - Pwn2Own 2018 Day One. https://www.thezdi.com/blog/2018/3/14/pwn2own-2018-results-from-day-one. Last Access: 2018-06-07.

[63] DHS Risk Steering Committee et al. DHS Risk Lexicon. *Washington, DC: The Department of Homeland Security*, 2010.

[64] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readac-

tor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (S&P)*, pages 763–780. IEEE, 2015.

[65] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *22nd ACM Conference on Computer and Communications Security (CCS)*, pages 243–255. ACM, 2015.

[66] Ryan Dahl. Node. js: evented I/O for v8 javascript. https://www. nodejs.org, 2012. Last Access: 2018-08-03.

[67] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. XIFER: a Software Diversity Tool against Code-Reuse Attacks. In *4th ACM International Workshop on Wireless of the Students for the Students (S3 2012)*, volume 174. Citeseer, 2012.

[68] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *13th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[69] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security)*, pages 401–416. USENIX Association, 2014.

[70] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend against Return-Oriented Programming Attacks. In *6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[71] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[72] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. Strict Virtual Call Integrity Checking for C++ Binaries. In *24th ACM Conference on Computer and Communications Security (CCS)*, pages 140–154. ACM, 2017.

[73] Stefan Esser. Exploiting the iOS kernel. https://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_Slides.pdf, 2011. Last Access: 2018-08-04.

[74] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *22nd ACM conference on Computer and Communications Security (CCS)*, pages 901–913. ACM, 2015.

[75] Andreas Follner and Eric Bodden. ROPocop–Dynamic Mitigation of Code-Reuse Attacks. *Journal of Information Security and Applications*, 29:16–26, 2016.

[76] Robert Gawlik and Thorsten Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *21st ACM conference on Computer and Communications Security (CCS)*, pages 396–405. ACM, 2014.

[77] Robert Gawlik and Thorsten Holz. SoK: Make JIT-Spray Great Again. In *12th USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2018.

[78] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *35th IEEE Symposium on Security and Privacy (S&P)*, pages 575–589. IEEE, 2014.

[79] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242. IEEE, 2018.

[80] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic Heap Layout Manipulation for Exploitation. In *27th USENIX Security Symposium (USENIX Security)*, pages 763–779. USENIX Association, 2018.

[81] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security)*, pages 177–192. USENIX Association, 2015.

[82] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Epressiveness of Non-Control Data Attacks. In *37th IEEE Symposium on Security and Privacy (S&P)*, pages 969–986. IEEE, 2016.

[83] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *12th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[84] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *2002 USENIX Annual Technical Conference*, pages 275–288, 2002.

[85] David Kaplan, Sagi Kedmi, Roee Hay, and Avi Dayan. Attacking the Linux PRNG On Android: Weaknesses in Seeding of Entropic Pools and Low Boot-

Time Entropy. In *8th USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[86] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of Android OpenSSL's Pseudo Random Number Generator. In *20th ACM Conference on Computer and Communications Security*, pages 659–668. ACM, 2013.

[87] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163. USENIX Association, 2014.

[88] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[89] Giorgi Maisuradze, Michael Backes, and Christian Rossow. What Cannot be Read, Cannot be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *25th USENIX Security Symposium (USENIX Security)*, pages 139–156. USENIX Association, 2016.

[90] Giorgi Maisuradze, Michael Backes, and Christian Rossow. Dachshund: Digging for and Securing Against (Non-) Blinded Constants in JIT Code. In *15th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2017.

[91] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *22nd ACM conference on Computer and communications security (CCS)*, pages 941–951. ACM, 2015.

[92] Matthew R Miller, Kenneth D Johnson, and Timothy William Burrell. Using Virtual Table Protections to Prevent the Exploitation of Object Corruption Vulnerabilities, 2014. US Patent 8,683,583.

[93] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *13th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[94] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. DROP THE ROP Fine-Grained Control-flow Integrity for the Linux Kernel. https://www.blackhat.com/docs/asia-17/materials/asia-17-Moreira-Drop-The-Rop-Fine-Grained-Control-Flow-Integrity-For-The-Linux-Kernel-wp.pdf. Last Access: 2018-08-03.

[95] George C Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *29th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 128–139. ACM, 2002.

[96] Ben Niu and Gang Tan. Modular Control-Flow Integrity. *ACM SIGPLAN Notices*, 49(6):577–587, 2014.

[97] Ben Niu and Gang Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *21st ACM Conference on Computer and Communications Security*, pages 1317–1328. ACM, 2014.

[98] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *22nd ACM Conference on Computer and Communications Security (CCS)*, pages 914–926. ACM, 2015.

[99] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating Return Oriented Programming Through Gadget-Less

Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[100] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *22nd USENIX Security Symposium (USENIX Security)*, pages 447–462. USENIX Association, 2013.

[101] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. MARX: Uncovering Class Hierarchies in C++ Programs. 15th Conference on Network and Distributed System Security Symposium (NDSS), The Internet Society, 2017.

[102] Mathias Payer, Antonio Barresi, and Thomas R Gross. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *12th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 144–164. Springer, 2015.

[103] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *13th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[104] Marco Prandini and Marco Ramilli. Return-Oriented Programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[105] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *25th USENIX Security Symposium (USENIX Security)*, pages 1–18. USENIX Association, 2016.

[106] Jesse Ruderman. Introducing jsfunfuzz. http://www. squarefree. com/2007/08/02/introducing-jsfunfuzz, 2007. Last Access: 2018-08-03.

[107] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, pages 448–459. ACM, 2016.

[108] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (S&P)*, pages 745–762. IEEE, 2015.

[109] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *20th USENIX Security Symposium (USENIX Security)*, pages 25–41. USENIX Association, 2011.

[110] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference*, pages 309–318, 2012.

[111] Fermin J Serna. The Info Leak Era on Software Exploitation. *Black Hat USA*, 2012.

[112] Alexey Sintsov. JIT-Spray Attacks & Advanced Shellcode. *HITBSecConf Amsterdam*, 2010.

[113] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *34th IEEE Symposium on Security and Privacy (S&P)*, pages 574–588. IEEE, 2013.

[114] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *14th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2016.

[115] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and Protecting Dynamic Code Generation. In *13th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[116] Alexander Sotirov. Heap Feng Shui in Javascript. *Black Hat Europe*, 2007, 2007.

[117] Jason Spielman. Deconstructing a Winning WebKit Pwn2Own Entry. https://www.thezdi.com/blog/2017/8/24/deconstructing-a-winningwebkit-pwn2own-entry. Last Access: 2018-06-07.

[118] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads. In *22nd ACM conference on Computer and communications security (CCS)*, pages 256–267. ACM, 2015.

[119] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security)*, pages 27–40. USENIX Association, 2014.

[120] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-Into-Libc Attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[121] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing Dynamic Binary Instrumentation Overhead. In *WBIA Workshop at ASPLOS*. Citeseer, 2006.

[122] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *22nd ACM conference on Computer and communications security (CCS)*, pages 927–940. ACM, 2015.

[123] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *24th ACM conference on Computer and communications security (CCS)*, pages 1675–1689. ACM, 2017.

[124] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *37th IEEE Symposium on Security and Privacy (S&P)*, pages 934–953. IEEE, 2016.

[125] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *19th ACM conference on Computer and communications security (CCS)*, pages 157–168. ACM, 2012.

[126] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. INSeRT: Protect Dynamic Code Generation against Spraying. In *2011 International Conference on Information Science and Technology (ICIST)*, pages 323–328. IEEE, 2011.

[127] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. No-Execute-After-Read: Preventing Code Disclosure in Commodity software. In *11th ACM ASIA Con-*

ference on Computer and Communications Security (AsiaCCS), pages 35–46. ACM, 2016.

[128] Rui Wu, Ping Chen, Bing Mao, and Li Xie. RIM: A Method to Defend from JIT Spraying Attack. In *2012 International Conference on Availability, Reliability and Security (ARES)*, pages 143–148. IEEE, 2012.

[129] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security)*, pages 781–797. USENIX Association, 2018.

[130] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From Collision to Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In *22nd ACM Conference on Computer and Communications Security (CCS)*, pages 414–425. ACM, 2015.

[131] Chao Zhang, Scott A Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. VTrust: Regaining Trust on Virtual Calls. In *14th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2016.

[132] Chao Zhang, Mehrdad Niknami, Kevin Zhijie Chen, Chengyu Song, Zhaofeng Chen, and Dawn Song. JITScope: Protecting web users from control-flow hijacking attacks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 567–575. IEEE, 2015.

[133] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *13th Conference on Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[134] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *34th IEEE Symposium on Security and Privacy (S&P)*, pages 559–573. IEEE, 2013.

Appendix A

LOOP Gadget Chain List

Table A.1: The Execution Flow of the LOOP Attack on Flash Player

Step	Gadget Name	Type	VTable Offset	Dispatch Offset	Argument
1	Flash!sub_307F7F20	T	0xA0	0x8	–
2	Flash!sub_30B7FFD0	T	0x8	0x10	–
3	Flash!sub_30280F9C	T	0x10	0x0	–
4	OLE32!CClientCallMgr::CPrivUnknown::QueryInterface	T	0x0	0x30	–
5	Flash!3022A484	AE	0x30	0x18	1 to 2
6	OLE32!CClassCache::CpUnkMoniker::BindToObject	T	0x18	0x0	–
7	OLE32!CAsyncUnknownMgr::CPrivUnknown::QueryInterface	T	0x0	0x58	–
8	Flash!sub_30241EF4	AE	0x58	0x18	2 to 4
9	OLE32!LEGACY_FRAME::GetInfo	T	0x18	0x20	–
10	Flash!sub_3022A470	T	0x20	0x10	–
11	OLEAUT32!DYN_TYPEBIND::AddRef	T	0x10	0x8	–
12	Flash!sub_307F56CC	I	0x08	–	–

Table A.2: The Execution Flow of the LOOP Attack on Internet Explorer

Step	Gadget Name	Type	VTable Offset	Dispatch Offset	Argument
1	IEFRAME!CInternetShortcut:: SetPath	T	0xA0	0x18	–
2	OLE32!CClassCache::CpUnkMoniker:: BindToObject	T	0x18	0x0	–
3	OLE32!CClientCallMgr::CPrivUnknown:: QueryInterface	T	0x0	0x30	–
4	MSHTML!CDXSimplifiedGeometrySink:: Close	T	0x30	0x48	–
5	IEFRAME!CIMTravelLogModel:: GetTravelLogEntryURL	T	0x48	0x20	–
6	MSHTML!CFrameContentHelper:: IsOpaque	T	0x20	0x50	–
7	IEFRAME!CIMTravelLogModel:: OnTravelLogChanged	AE	0x50	0x18	1 to 3
8	OLE32!CClassCache::CpUnkMoniker:: BindToObject	T	0x18	0x0	–
9	OLE32!CClientCallMgr::CPrivUnknown:: QueryInterface	T	0x0	0x30	–
10	MSHTML!CDXSimplifiedGeometrySink:: Close	T	0x30	0x48	–
11	IEFRAME!CIMTravelLogModel:: GetTravelLogEntryURL	T	0x48	0x20	–
12	MSHTML!CSVGFilterElement:: GetTargetBounds	T	0x20	0x38	–
13	MSHTML!CPublicTravelLog::GetCount	AE	0x38	0x38	3 to 4
14	MSHTML!CDXSimplifiedGeometrySink:: SetFillMode	T	0x38	0x18	–
15	MSHTML!CursorConsumer::Bind	T	0x18	0x10	–
16	MSHTML!CLayout:: AddLayoutTaskOwnerRef	T	0x10	0x60	–
17	MSHTML!Ptls6::CLsBlockObject::Display	I	0x60	–	–

List of Figures